NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMMERSIVE ARTICULATION OF THE HUMAN UPPER BODY IN A VIRTUAL ENVIRONMENT

by

Paul F. Skopowski

December 1996

Thesis Co-Advisors: Robert McGhee
John S. Falby

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Master's Thesis

4. TITLE AND SUBTITLE
Immersive Articulation of the Human Upper Body in a Virtual Environment

5. FUNDING NUMBERS

6. AUTHOR(S)
Skopowski, Paul F.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis addresses the problem that virtual environments (VE's) do not possess a practical, intuitive, and comfortable interface that allows a user to control a virtual human's movements in real-time. Such a device would give the user the feeling of being immersed in the virtual world, greatly expanding the usability of today's virtual environments.

The approach was to develop an interface for the upper body, since it is through this part of users' anatomy that they interact most with their environment. Lower body motion can be more easily scripted. Implementation includes construction of a kinematic model of the upper body. The model is then manipulated in real-time with inputs from electromagnetic motion tracking sensors placed on the user.

Research resulted in an interface that is easy to use and allows its user limited interaction with a VE. The device takes approximately one sixth the time to don and calibrate as do mechanical interfaces with similar capability. It tracks thirteen degrees of freedom. Upper body position is tracked, allowing the users to move through the VE. Users can orient their upper body and control the movements of one arm. Uncorrected position data from two trackers was used to generate clavicle joint angles. Difficulty in controlling figure motion indicates that the sensors used lack sufficient registration for this purpose. Therefore, the interface software uses only orientation data for computing joint angles.

14. SUBJECT TERMS
human interface, virtual environment, articulated humans, human modeling, motion trackers, electromagnetic sensors, Polhemus, kinematics, NPSNET

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited.

IMMERSIVE ARTICULATION OF THE HUMAN UPPER BODY IN A VIRTUAL ENVIRONMENT

Paul F. Skopowski
Major, United States Marine Corps
B.S.S.E. , United States Naval Academy, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1996

Author: _____
Paul F. Skopowski

Approved by: _____
Robert B. McGhee, Thesis Co-Advisor

_____
John S. Falby, Thesis Co-Advisor

_____
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

This thesis addresses the problem that virtual environments (VE's) do not possess a practical, intuitive, and comfortable interface that allows a user to control a virtual human's movements in real-time. Such a device would give the user the feeling of being immersed in the virtual world, greatly expanding the usability of today's virtual environments.

The approach was to develop an interface for the upper body, since it is through this part of users' anatomy that they interact most with their environment. Lower body motion can be more easily scripted. Implementation includes construction of a kinematic model of the upper body. The model is then manipulated in real-time with inputs from electromagnetic motion tracking sensors placed on the user.

Research resulted in an interface that is easy to use and allows its user limited interaction with a VE. The device takes approximately one sixth the time to don and calibrate as do mechanical interfaces with similar capability. It tracks thirteen degrees of freedom. Upper body position is tracked, allowing the users to move through the VE. Users can orient their upper body and control the movements of one arm. Uncorrected position data from two trackers was used to generate clavicle joint angles. Difficulty in controlling figure motion indicates that the sensors used lack sufficient registration for this purpose. Therefore, the interface software uses only orientation data for computing joint angles.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My sincerest thanks to all those who helped make this thesis possible. Many thanks to Captain Marianne Waldrop for getting me interested in this work and helping to kick-start this research by providing her work and insights. I would also like to thank Dr. Scott McMillan. Without the benefit of his research and patient instruction into the finer aspects of kinematic modeling, this thesis would not have progressed as far as it did. Special thanks go to my two thesis advisors. Their seemingly endless energy and devotion to the needs of their students is nothing short of amazing. To John Falby I owe much. His many hours of instruction over the past two years have brought me a long way in understanding the science of computers. He's a real friend who is always willing to drop what he's doing to help others with a problem. To Dr. Robert McGhee, I have not known a finer mentor. His vast experience and unbounded enthusiasm, patience, and encouragement have made working with him a true delight. My sincerest gratitude to the rest of the students and staff of the Computer Science Department. They helped in numerous ways, both big and small. I have not worked with a finer group of people. Finally, but most importantly, thank you to my wife Theresa. Without her unwavering support, love and devotion through the years, this and many other life achievements would not have come to pass.

I. INTRODUCTION

A. MOTIVATION

There is a growing requirement for realistic virtual environments (VE) in which humans can interact. VE's offer a safe, economical and efficient means to explore or train in otherwise hazardous or inaccessible environments. Uses for VE's are numerous and span disciplines that include engineering, science, education, entertainment, military, law enforcement and medicine. Recent advances in computer and motion sensor technologies have made it feasible to insert humans into the VE and control their movements in real-time. Presently, however, interfaces to manipulate virtual humans are not well developed and are only available inside the research community. An urgent need exists to provide a practical, intuitive, and comfortable interface that allows its human user to feel as if he is immersed in the virtual world. In particular, since humans largely interact with their environments through their hands, an acceptable interface for the upper body is critical.

B. GOALS

The purpose of this thesis is to use technologies currently available to provide the user with a practical interface for manipulating the upper body of a human icon inserted in a VE. The approach is to place motion sensors on the user and have the user's movements tracked and then replicated in the virtual world. With this in mind, there are three major goals for the research of this thesis. First, the interface should be effective in driving realistic and reasonably accurate movement of the virtual human. For example, if the user touches his right shoulder with his left finger tips, the virtual human should move its joints in the same manner. Ideally, when the motion is complete, the joint angles of the icon match those of the user. Also, the finger tips of the icon should be touching its shoulder and not hanging in space some distance from the shoulder. Second, the interface should be efficient

enough to ensure that all actions commanded occur in real-time. The action is represented graphically as a smooth flow of motion. An added benefit of an efficient interface concerns its possible future use in large scale networked VE's where time delays are more critical. Third, the interface must be intuitive and easy to use. The user can quickly learn how the icon's movements correlate to his own. Additionally, the system must be relatively easy to calibrate and the user reasonably unencumbered while wearing the sensors. Collectively, the success in achieving these goals is determined by whether a user can complete a particular set of tasks or training in the virtual world [ZYDA92].

C. ORGANIZATION

Chapter II of this thesis provides background information and reviews previous work related to the area of interactive human interfaces for virtual environments. Chapter III provides an overview of kinematics modeling and discusses the development of the specific kinematics model used to effect the interface. The last part of this chapter discusses a prototyping tool used to implement the model. The tool is written in C++ using OpenGL graphics libraries. Chapter IV contains a description of the motion tracking equipment, the inverse kinematics implementation, and the software used to interface the sensors with the prototype tool discussed previously. Chapter V provides a discussion of the development and implementation of the calibration routines used. Chapter VI presents results obtained from this research. The last chapter, Chapter VII, provides some conclusions and discusses recommendations for future enhancements and research.

II. BACKGROUND

The speed and power of today's computers have made it possible to create realistic virtual worlds that can be populated with dynamic entities portrayed in real-time. Presently, articulated entity motion is primarily scripted and non-interactive [PRAT95]. One of the greatest challenges facing researchers concerns the insertion of individual humans into these interactive environments [WALD95].

In meeting this challenge, researches are faced with several tasks. These include: 1) creating a model of the human body with the desired level of detail to be used in the virtual world, 2) defining the level of control desired and types of user inputs required to manipulate the model, and 3) providing the inputs to the model in an acceptable form and timely manner. Tasks 2 and 3 are the essence of an interface between the computer generated model and user. The desire is to control most, if not all, the degrees of freedom represented in the model. (The level of detail of the model used is discussed below and in Chapter III.) An ideal form of user inputs is the user's own natural movements. This provides as intuitive an interface as possible. The goal is to make the user feel as if he *is* the figure being animated, thus giving him the feel of being immersed in the virtual environment. The input devices are motion sensors attached to the user and the associated software. The remainder of this chapter discusses the modeling of humans, tracking of humans, and work that has been done in putting the two together in an interface.

A. MODELING HUMANS

First, it is necessary to develop an acceptable model of the human body. Modeling provides a means to represent the salient characteristics of a complex system such as the human body. The idea is that an appropriate representation of the human body (the model) will be used to help efficiently describe realistic human motion in three dimensional space.

The question is, what level of detail is required in the model to effectively represent the human in the virtual world? This depends on the effects one wishes to achieve by inserting the human into the virtual environment and the techniques that can be used together with the model to achieve the desired results. The terms *manipulation, animation,* and *simulation* describe methods that use models to create visual effects. Before discussing the methods that have been used to model humans, a discussion of these terms is warranted.

1. Manipulation, Animation, and Simulation

Manipulation generally involves the movement of objects in direct response to the actions of the user. Manipulation is inherently real-time and control of the figure lies largely with the user, though some constraints in the model may prevent certain unrealistic movements. Historically, manipulation has been used to reposition human figures into various static postures [BADL93a].

If the goal of manipulation is to direct a movement, then the goal of animation is to describe or choreograph motion [BADL93a]. Animation generally has an artistic quality about it. Success is often measured in terms of how well the motion was expressed. Animation techniques are generally time consuming and executed off-line since higher levels of accuracy and control are desired.

Simulation can be defined as the process where one system's behavior (the original object) can be predicted or extrapolated by observing the behavior of another system (the model) [CANT95]. With respect to motion, simulation has been described as automated animation [BADL93a]. In this case, the user describes an input ahead of time and the system generates the motion. The input is usually a goal and possibly some rules for making decisions. Control of the model can occur at a higher level. The resultant motion is no longer entirely in the hands of the user but can be heavily impacted by definitions of the model. Simulation, however, can result in very realistic movements. This form of

animation is unique to computers and can be a powerful tool in the field of scientific visualization [WATT92] and the design of feedback control systems for legged robots [MCMI96a]. The drawbacks, however, are that the models used are more complex and computations associated with them incur more overhead.

Obviously, the distinction between manipulation, animation, and simulation is not black and white and none of these terms is mutually exclusive of the others. For example, by manipulating all the parts of a figure simultaneously, one can in fact animate it. Also, placing joint angle limits on an otherwise simple graphical model to limit manipulation could be considered the beginnings of a crude simulation. The terms are helpful, however, for understanding the relationship between a model and its primary uses. With this in mind, it is appropriate to consider modeling methods that have been used in the past.

2. Modeling Methods

There are several methods that can be used to model a human. These include graphic, kinematic, and dynamic modeling methods. Classifying a model as a given type is largely a matter of its prevalent features, although a clear distinction between types may not be possible or necessary. The various modeling methods can actually be thought of as existing at points on a line that represents a continuum of options and features. Graphic models would exist at the far left of the spectrum and represent the simplest form of model. Dynamic, or physically based models would exist at the far right of the spectrum. Finally, kinematic models would exist somewhere in-between.

a. Graphic Models

Graphic models are often used in conjunction with animation systems and commercial animation packages. These systems usually provide a means of attaching objects to each other. The user can construct hierarchies of objects. Manipulation of a parent object necessarily results in movement of its child object, though there is no real

notion of articulation. Simply put, the origin of the child object is relative to that of the parent [BADL93a]. A graphic model of a human could be created by simply attaching various three dimensional shapes together as needed. Needless to say, numerous graphic models of humans have been created with varying levels of detail.

b. Kinematic Models

Kinematics is the science of motion which treats motion independent of the underlying forces that cause it [CRAI89]. It includes position, velocity and acceleration and relates geometric and time considerations. Kinematics models are often used in the field of robotics. Research in this field has resulted in two standardized and well known kinematic notations. These are the Danevit and Hartenberg (DH) and the Modified Danevit and Hartenberg (MDH) notations [DAVI93]. The kinematic model using these notations specifically describes physical links connected by either revolute or prismatic joints. These models can be highly efficient at describing articulated rigid bodies [MCMI94].

Kinematic computations generally fall into two categories: forward kinematics and inverse kinematics. In forward kinematics, the position and orientation of the last link (called end-effector) of a series of connected links is calculated given the joint angles associated with each joint in the chain. The motion of the end-effector is determined by the accumulation of transformations calculated from a base link down the entire series of links to that end effector [WATT92].

Inverse kinematics is generally not as straightforward as forward kinematics. Inverse kinematics entails calculating joint angles given the position and orientation of an end-effector and sometimes intermediate links. In inverse kinematics, as the number of links in the chain increase, the amount of link position and orientation information required to unambiguously determine joint angles increases. If not enough information is provided, the system is said to be *redundant*. In such a case, additional

constraints or heuristics must be applied to the system to achieve a unique solution. Examples of constraints include such things as energy minimization and momentum conservation [WATT92]. Methods for solving inverse kinematics tend to be unique to each specific case.

Both forward and inverse kinematics have been used in animation. The animator can create a 'kinematic skeleton', or model, and manipulate it using either method. When the desired movement is obtained the animator can then, if required, 'cloth' the skeleton [WATT92]. The advantage of using forward kinematics to manipulate such a structure is that it affords the animator more control over the figure's positions. Its disadvantage is that it is counterintuitive and complicated to use in practice. It is difficult to specify directly all the joint angles needed to define an exact posture of a complex figure such as the human. Use of inverse kinematics can provide relief from having to specify all joint angles, though some control may be forfeit. Disadvantages of inverse kinematics include possible ambiguities and the complexity of computations. The resulting additional overhead may be critical if the application is intended to run in real-time.

Because it offers an efficient, standardized and well-developed representation of the motion of articulated bodies, a kinematic model is used in this research. Chapter III discusses the notation, methodology and details of the kinematic model developed as part of this research.

c. Dynamic Models

Dynamics is the study of motion and the forces effecting that motion. The dynamic model of a human would describe body position and movement largely as a result of the underlying forces which the neuromuscular system exerts on the body together with the force of gravity. A rich simulation of the human body would necessarily include some

aspects of dynamic modeling. Human traits such as weight, strength, and balance require the consideration of the underlying forces that cause them.

There are a number of methods available to simulate the dynamics of articulated bodies. These methods can take one of two opposing views of the world in their approach.

In the first case the view of the world is one of objects and constraints. In this view, human limb segments are treated as individual objects and are kept in place by heavily weighted constraints (forces) that join them. There is no real notion of articulation. For complex objects, this approach can be computationally time-consuming [KOOZ83].

In the second case, the world can contain articulated bodies. These bodies maintain a tree structure with a base link just as in the kinematics approach. The effects of forces can be propagated down a chain of links from the base to affect the body as a whole. Two of the most efficient methods for achieving these computations are the Composite Rigid Body (CRB) method and the Articulated Body (AB) method [MCMI95]. It has been shown that the AB method grows in computation linearly with the number of degrees of freedom modeled, whereas the CRB method grows as the cube of the number of degrees of freedom modeled [MCMI95]. Real-time simulation of complex articulated bodies (24 degree of freedom (DOF) robots) has been achieved using the AB method [MCMI95, MCMI96a]. By contrast, an elegant dynamic model of a human (30 DOF) that does not use this method has yet to achieve a real-time capability [HODG95]. This is an active area of research and one that could result in real-time dynamic models of humans in the near future.

B. TRACKING HUMANS

Tracking of humans is a basic requirement of almost all VE systems. The focus of this research is on the human upper body, exclusive of the head. Focus is on the upper body since humans interact with their environment largely through this portion of their anatomy.

Routine lower body movements such as standing, walking or running can be easily scripted or controlled by an appropriate stepping algorithm [GUBI74, KWAK90]. Further, head tracking requirements are highly application dependent and involve close correlation with the type of visual display being provided to the user. Systems exist and are available explicitly for this purpose. Interfaces for the upper body are less well developed.

Requirements levied on a system to track the upper body will largely be driven by requirements to track the hands. This is due to the fact that the range and speed of motion of the hands is greater than that of other parts of the upper body. Tracking devices whose sampling rates are fast enough to track the hands will be fast enough to track other parts of the body. By assuming 5 Hz as the defining frequency for hand motion and requiring 20 times oversampling for sensor noise, [DURL95] estimates a sampling rate of 100 Hz for tracking the human hand. It is desirable then that any device chosen to track the upper body have a sample rate of at least 100 Hz.

There are several methods that can be used to track the human upper body. These include mechanical, electromagnetic, acoustic, optical, and inertial methods. Correspondingly, there is a variety of trackers either under development or offered commercially. The research community is in a constant search for effective three-dimensional motion trackers that are affordable and easy to use [WALD95]. As a result, measures of performance for comparing trackers have been developed and studies of trackers have been conducted [MEYE92, DURL95, FREY96]. Once the type of tracker to be used is determined, how many are needed, where they should be physically placed, and how they will be calibrated must be considered. The following paragraphs discuss how the various types of motion trackers can be compared.

1. Measuring Motion Tracker Performance

Determining which tracker to use requires the use of standard measures performance. Currently, there is a lack of agreement on performance specifications and how they should be measured [DURL95]. Work in this area is needed before more quantifiable comparisons can be made. In the mean time, [MEYE92] suggests motion trackers be evaluated in more general terms by the following certain key measures, namely; (1) *resolution and accuracy,* (2) *responsiveness,* (3) *robustness,* (4) *registration,* and (5) *sociability*. *Resolution* is defined by the smallest change that can be detected by the system. *Accuracy* is considered the range over which the measured quantity is correct. Accuracy would include a sensor's drift; i.e., the tendency of its output to change without any change in input. *Responsiveness* is a measure of the quickness with which new information is provided. It is determined by *sample rate, data rate, update rate* and *latency,* with *latency* being the most critical factor. *Latency* is sometimes called *lag*. It is the delay between the movement of the sensed object and report of the new position and orientation. *Robustness* is a measure of the tracker's effectiveness in the presence of noise or other signal interference (including shadowing) in the operating environment. *Registration* is the correspondence between a unit's actual position and orientation and its reported position and orientation over the domain of the working volume. Lastly, *sociability* is a measure of how well the tracker interfaces to track multiple objects in the same environment and the *range of operation* at which it can function. *Range of operation* is sometimes referred to as *working volume*.

In addition to considering these performance factors, one might consider availability, cost, and ease of use before actually selecting a position tracker. Certainly availability or cost can put a system out of reach. If the virtual reality application associated with the sensors is intended for general use, then it is desirable that the system be easy to

set up and use. Additionally, one would like the user's movements to be unincumbered since a free range of motion is necessary to enhance the illusion of being immersed in the virtual environment.

2. Types of Motion Trackers

There are many different motion trackers being developed and marketed. A brief overview of types of trackers and their pro's and con's follows. This section concludes with a discussion of why a specific tracker type was selected for this research.

a. Mechanical

Mechanical trackers measure change in position and orientation by physically connecting the object being tracked to a point of reference with jointed linkages. In the case of a human, the point of reference can either be another part of the human body or a fixed surface near the human. Thus, these trackers can be separated into two basic types, body-based (exoskeletal systems) and ground-based systems [DURL95]. Body-based systems are used to track the user's joint angles or end-effector positions relative to some other part of the body. Ground-based systems are attached to some surface near the user. Generally, the user grasps an implement whose position and orientation are tracked.

The fact that mechanical trackers are a system of physical linkages attached to the body or constantly held makes them cumbersome. This is an undesirable trait if the goal is to give the user the freedom of motion and activity associated with being totally immersed in a virtual environment. This feature, however, makes mechanical trackers an excellent choice for haptic (force-feedback) devices, since they are rigidly mounted to the user and/or a nearby surface [DURL95]. Also, mechanical trackers tend to be accurate, responsive, and robust. On the other hand, they have poor sociability [MEYE92] and can be difficult and time consuming to calibrate [DURL95, PRAT95].

b. Electromagnetic

Electromagnetic trackers are trackers that utilize the electromagnetic spectrum. *Electromagnetic spectrum* is defined in a narrow sense to mean radio and microwave frequencies. The two methods considered here include electromagnetic and spread-spectrum ranging techniques.

Electromagnetic position trackers work on the principle that a magnetic field induces voltage in a coil. Typically, these systems comprise a transmitter and receiver. The emitter generates three mutually perpendicular electromagnetic fields if the coil is in motion or the field is changing. The receiver is comprised of three orthogonal coils, each generating its own voltage in the presence of the electromagnetic fields for a total of nine voltages. The voltages generated are used to determine the sensors orientation. Voltage strengths of the three transmitted signals are used to determine the location of the receiver [MEYE92].

Electromagnetic trackers have been commercially available for some time and are reasonably inexpensive and easy to use. Not surprisingly, they are by far the most prevalent in use today. They tend to have good accuracy in a small working space, with accuracy trailing off as distances from the transmitter increase. Robustness is adversely effected by sensitivity to ferromagnetic objects in the vicinity, with AC-based trackers being more susceptible than DC-based trackers. AC-based systems tend to generate eddy currents in metallic objects which then cause their own electromagnetic interference. Adding power to the transmitter to increase the working volume can increase noise. Both systems are adversely impacted by noise from power sources. Responsiveness is poor compared to other methods. Moreover, this feature adversely impacted by design provisions that enhance robustness, particularly in AC-based systems. In the past, use of noise filtering techniques caused additional lag problems [MEYE92]. Magnetic systems, however, are unaffected by non-ferromagnetic occlusions, a big plus. Sociability is best in

an environment without ferromagnetic occlusions, but limited due to a small range of operation. Still, these systems can be very effective at shorter ranges.

A second method recently proposed involves use of spread-spectrum ranging techniques [BIBL95]. This technique uses the measured time of arrival of electromagnetic pulses to determine range from a set of fixed transmitters (or receivers). The concept is similar to that of the Global Positioning System used for navigation. A minimum of three fixed transmitters would be required to determine position via triangulation, plus a fourth transmitter to ensure time can be accurately computed by the receiver [LOGS92]. Transmitted signals all occupy the same wide bandwidth and utilize code division multiple access (CDMA) to preclude mutual interference.

Though these systems are not currently available, it is likely that they will be soon. Recent developmental efforts have shown the concept to be technically feasible [ADVA96]. Work performed by Advanced Position Systems, Inc., under contract to the Naval Research Laboratory, Washington, D.C., has shown that head tracking with accuracy in the millimeter range is possible. Sampling was conducted at 1000 Hz. The accuracy and responsiveness shown by this type of system will remove many of the limitations of current electromagnetic systems. Additionally, if spread spectrum transmission techniques are used, they are very robust in the presence of noise. However, they are not immune to ferromagnetic occlusions. Perhaps most significantly, sociability of this technique would be excellent due to longer range of operation and an ability to track a large number of targets simultaneously [BIBL95].

c. Acoustic

Acoustic systems use ultrasonic devices to track objects via one of two methods. The time-of-flight (TOF) method uses the known speed of sound through air to calculate distances between transmitter and receiver. Once these distances are known,

triangulation between several receivers and one transmitter (or vice-versa) can be used to determine position. Orientation is determined by tracking position at three locations on the same object. The second method is called the phase-coherence (PC) method. This method senses phase difference between transmitted and received signals and converts a change in phase to a change in position. Objects that move farther than half a wavelength in one update period will induce tracking error. Because of this, small errors in position determination can result in larger errors over time [FREY96, LIPM90].

Performance of these two systems can vary greatly in an open room environment. Phase-coherent systems enjoy many benefits over time-of-flight systems due to much higher data rates [MEYE92]. If the range is small, both systems offer good accuracy, responsiveness, and robustness. As range increases, data rates for time-of-flight systems decrease, causing responsiveness and robustness to decrease. Both systems suffer severe effects from occlusion. Sociability of phase-coherent systems, however, is better than that of time-of-flight systems due to larger working volumes [MEYE92].

Acoustic systems are relatively inexpensive. They offer better range of operation than magnetic systems but can suffer severe effects from shadowing that can occur between tracked body parts or other objects.

d. Optical

There are more types of optical trackers than any other position tracker. Generally, these can be broken down into those that use unstructured light (usually infrared or ambient light) and those that use structured light (i.e. lasers).

Optical trackers that use unstructured light can place the sensor on the tracked object and emitters at fixed locations around the tracked object (inside-out systems) or vice-versa (outside-in). These systems more typically track objects by placing a set of cameras (or other light sensitive devices) at fixed locations around the operating volume.

Targets may be marked with actively emitting light sources (often infrared diodes) [MCGH79, KOOZ80] or they may be passively tracked. Triangulation is then used to determine position. Distinguishing common reference points in a scene between cameras can be a problem, especially for systems that don't use marking. Use of a short focal length to enable the camera to view a larger volume causes reduced accuracy at longer ranges.

Pattern recognition systems are image-based systems that determine position by comparing known patterns against sensed patterns. These systems become less accurate as range increases since the virtual image size of the object decreases. Additionally, they require complex algorithms to interpret scene content [MEYE92].

Structured light systems use lasers to scan a scene. Beam forming optics are used to create a plane of coherent light whose reflections are then captured by a two-dimensional camera. The intersection of the known plane and the line of sight from the camera are used to determine three-dimensional coordinates. Another common method uses laser spot scanning of the scene. In this method either all or only portions of the scene may be scanned for data [DURL95].

Two of the most prevalent techniques that use lasers include laser radar and laser interferometry techniques. Laser radar works in the same way as acoustic ranging techniques, except that much higher data rates are possible. Laser radar techniques include time-of-flight and phase shift techniques. By scanning the entire scene, a three-dimensional picture of the scene can be generated. Laser radar techniques are more appropriate for longer distances than laser techniques that use triangulation [DURL95].

The second technique, laser interferometry, uses a steered laser beam to track a reflector on the object being tracked. Phase-shift ranging and angular information from the steered system are used to determine position. A second method uses several lasers to track the reflector from different fixed positions and range information only. In this case, the intersection of spheres whose radii are determined from the range information

and whose centers are located at each laser determines the location of the point being tracked. The problem with these techniques is that they provide only incremental displacement data and loss of signal via shadowing can be cause for recalibration [DURL95].

Though there are numerous types of optical tracking systems, some general comments about their performance can be made. Most optical systems must inherently trade-off between accuracy and range of operation. Additionally, all optical systems can be impeded by the effects of shadowing [FREY96]. To get around these problems, designers often develop multiple emitter/sensor architectures that are complex and expensive. Conversely, however, these systems can have very high resolution and accuracy, especially structured light systems. They tend to be very responsive due to high data rates, and so are well suited to real-time applications [MEYE92]. Sociability is system dependent, but can be limited by range of operation and shadowing considerations.

e. Inertial

Inertial trackers that utilize small micro-machined linear accelerometers and angular rate sensors can create an "artificial vestibular system" [BACH96a] for tracking orientation. An angular rate sensor operates by using the differential combination of the outputs of two vibrating linear accelerometers or by sensing "Coreolis" torques at the base of a vibrating tuning fork [SYSTRO]. Angular rate sensor drift is compensated for by using linear accelerometers together with the earth's gravitation field to sense angular orientation. Since angular rate sensors are accurate for high frequencies and linear rate sensors are accurate for low frequencies, a cross-over filter can be used to combine the inputs from each. Additionally, the earth's magnetic field may be used to compensate for drift in azimuth. Three of each type of accelerometer (linear and angular) orthogonally

oriented are needed to fully describe an object's position and orientation in space [FREY96].

Several companies have begun manufacturing small accelerometers but devices small enough for use in tracking angular orientation of human body parts are just now becoming available [INTE96]. These systems hold much promise for future application to the body tracking problem [FREY96]. Larger systems have shown that accuracy, resolution, response, robustness and registration requirements for human body tracking can be met by this technology [BACH96a]. Although the technology is currently expensive, it is expected that costs will come down as devices are marketed. Perhaps the greatest benefit of this type of system, however, relates to its sociability. Whereas electromagnetic, acoustic and optic devices all require emissions from a source to track objects, inertial trackers are sourceless. This precludes the inevitable disastrous effects of occlusions and noise, with the exception that some form of wireless transmission will be required to pass data from the tracked object.

Table 1 provides a summary of the position tracker capabilities discussed [BIBL95, DURL95, FREY96, MEYE92]. Of the types of trackers listed, inertial and spread-spectrum systems were not commercially available during the research of this thesis. Additionally, the cost and complexity of optical systems has precluded their consideration for use in this research. The remaining three technologies include mechanical, magnetic and acoustic trackers. Mechanical techniques for tracking the upper body have been implemented and have been shown to be cumbersome and difficult to use. Acoustic trackers can provide potentially excellent accuracy and resolution together with a greater range of operation than magnetic trackers. However, acoustic trackers can suffer severe effects from shadowing of one body part by another, whereas magnetic trackers do not. This is a key factor in selecting magnetic trackers for use in this research, despite this technology's limitations. It also indicates the potential of other technologies that don't

Property	Mechanical	Electromagnetic	Acoustic	Optical	Inertial
Accuracy & Resolution	Good	M: Moderate SS: Excellent	TOF: Good PC: Excellent	Excellent	Good
Responsiveness	Good	M: Poor SS: Good	TOF: Moderate PC: Good	Excellent	Good
Robustness	Excellent	M: Moderate SS: Good	TOF: Poor PC: Good	System Dependant	Excellent
Registration	Unknown	Unknown	Unknown	Unknown	Good
Sociability	Poor	M: Poor SS: Excellent	System Dependent	System Dependent	Excellent
Ease of Use	Poor	Good	Good	Good	Good
Cost	Acceptable	M: Inexpensive SS: Expensive	Inexpensive	Expensive	Expensive
Availability	Yes	M: Yes SS: No	Yes	Yes	No

M: magnetic field method
SS: spread spectrum method
TOF: time of flight method
PC: phase-coherence method

Table 1. Tracking Technologies Compared

suffer the effects of shadowing such as spread-spectrum ranging and inertial trackers. These technologies have potential for providing much needed increases in accuracy, response and range of operation over magnetic systems available now.

3. Motion Tracker Placement

Once a specific motion tracker is selected, placement of the trackers requires consideration. Specification of the number and placement of trackers to be used on the human body depends on a number of related factors. Some of these factors as they relate to electromagnetic trackers are identified here, though similar considerations apply to other types of trackers.

a. Number of Motion Trackers

In order to unambiguously specify a single rigid body's position and orientation, a single six degree of freedom (DOF) tracker is required. The tracker provides the x, y, and z coordinates and roll, pitch and yaw of the tracker relative to some fixed frame of reference. Alternatively, one could use three three DOF trackers that provide x, y, and z position only and place them on different locations of the same body. Suppose the rigid body is attached to another and the attachment point provides three degrees of freedom (such as a ball and socket joint). If the x, y, and z coordinates of this attachment point are known, then either a single three DOF orientation tracker or two three DOF position trackers (not colinear with the attachment point) placed on the body will be sufficient to specify all six DOF of the body. This second example can be considered a simple articulated body.

For articulated bodies, determining the required number of trackers needed is dependent on the DOF's inherent in the body. As seen from the previous example, it can also depend on the amount and type of DOF data provided by the tracker. For now, only six DOF trackers will be considered. Take for example a six DOF robot arm with the end of its base link fixed at a known position. All linkage position and orientations can be specified by using one six DOF tracker that measures position and orientation and is placed on the end-effector. Inverse kinematics is used to determine the joint angles of the links. If the robot arm were to have seven DOF's as in the human arm, then a single six DOF tracker at the end-effector would not be sufficient to reproduce all the joint angles associated with the system. In such a case an infinite number of solutions for one or more joint angles results.[1] In general, for articulated bodies that are tracked by six DOF trackers, the number

1. For a more in depth discussion of the solvability of inverse kinematics problems see [CRAI89].

of trackers required can be determined by dividing the number of DOF's in the body by six and adding one if any remainder results.

The terms *under specified*, *completely specified* and *directly specified* are introduced to help provide a framework of understanding. The term *under specified* is used when the number of trackers associated with the system is not sufficient to unambiguously determine all the joint angles of the articulated body. In this case, inverse kinematics will result in an infinite set of possible linkage positions. Additional constraints on the system must be identified before a unique solution will result. The term *completely specified* is used when the number of trackers associated with the system is sufficient to unambiguously determine all joint angles of the articulated body. In robotics, the term *fully specified* has the same meaning as completely specified. The term *directly specified* is used when an articulated body is completely specified and all physical links' joint angles can be *directly measured*. *Directly measured* means that data from one or more trackers on the physical link can be used to determine the joint angles to within a constant transformation matrix.

In designing an interface, selection of the type of system (under, completely or directly specified) can have an impact on performance. For a real-time application, the goal is speed. Ideally, one would like to provide just enough data to completely specify the system while avoiding the computational overhead associated with using complex inverse kinematics. A completely specified system avoids the computational overhead associated with implementing algorithms that enforce realistic constraints on the system. A directly specified system ensures that only the simplest of inverse kinematics are used to determine all limb segment orientations. The trade-offs associated with a directly specified system include the cost of additional hardware and a more encumbered user.

If it is desired to directly measure the joint angles associated with a link, then a single six DOF tracker attached to the link will suffice. If the x, y, and z coordinates of the joint attaching the link to the rest of the body are known, then a single three DOF

orientation tracker is all that is required. From this, it is easy to see that by knowing or tracking the position of the base link of an articulated body all that is needed to fully specify the system is one three DOF orientation tracker on each link. Table 2 shows the number of sensors required for complete and direct specification of the human arm.

Joint	Arm Segment	Joint DOF	Cumulative DOF[a]	# Sensors Complete Specification[b]	# Sensors Direct Specification[c]
Shoulder	Upper Arm	3	3	1	1
Elbow	Forearm	1	4	1	2
Wrist	Hand	3	7	2	3

Table 2. Joint DOF's and Sensor Requirements [WALD95]

a. assumes coordinates of shoulder are known
b. elbow and wrist joint sensor numbers reflect use of six DOF sensors
c. only three DOF orientation trackers required

b. Physical Placement of Trackers

In a general sense, the possible placement of trackers was alluded to in the previous section. If it is desired to create a directly specified system for the anticipated speed it offers, a six DOF tracker can be placed on the base link and three DOF orientation trackers on each additional link. With this information, the figure's complete state can be specified. There are other considerations, however, for placing the trackers.

There are several problems associated with using trackers to infer joint angles. The first concerns soft tissue and the potential for relative motion between the tracker and the limb segment [DURL95]. For example, suppose a tracker is placed on the upper arm near the shoulder to measure three DOF motion. It is unlikely that this tracker will sense the roll of the upper arm accurately, since the tissue it is attached or strapped to remains relatively stationary in relation to the underlying bone structure. A better place to sense shoulder roll would be near the elbow or possibly even on the forearm near the elbow.

A second problem associated with inferring joint angles involves the fact that human joints are not perfect hinge or spherical joints [DURL95]. In fact, their axes of rotation move with the joint angle. In some cases they aren't revolute joints at all. For example, the forearm can be rolled yet the elbow is a single DOF hinge joint. This is because the bones of the forearm form a closed chain of links with two bones twisting around each other to allow the wrist to turn. In the first case, where human joints approximate revolute joints, the effects of not using a model and calibration scheme to account for this are within the expected resolution of today's magnetic trackers. In the second case, a simplified model can still be used dependent on the level of detail desired in the application. It is important, however, to position the tracker where the DOF's modeled can actually be sensed.

So far, implementations which rely exclusively on three DOF position trackers have not been discussed. In fact, some tracking systems only provide positional data from which orientation must be derived. It was already shown how two position trackers can replace one orientation tracker on a single link. This is, however, an increase in the number of sensors a user must wear. As mentioned earlier, considerations for placement are similar to those for the orientation tracker with the added constraint that sensors not be colinear or near colinear with each other and the joint.

One suggestion has been to place one three DOF position tracker at each joint of the human [BIBL95]. Placing three DOF position trackers at the joints is less problematic with regards to soft tissue, since joints generally have less soft tissue on them. Problems arise, however, with regards to determining methods of attachment at the joints that are secure, yet unencumbering to the user. The sensors can be offset from the joints to mitigate these effects, though calibration of the offset will be required. This system, however, is not directly specified and requires more complex inverse kinematics to

determine the orientation of each limb. At least two additional position trackers will be needed on the end-effectors (hands, head, etc.) to make the system completely specified.

4. Motion Tracker Calibration

Once the trackers are placed on the tracked object or limb, they must be calibrated. The calibration process is used to determine the relationship between the tracker's frame of reference and the tracked object's frame of reference. Once the transformation matrix between the tracker and object is known, then data reported by the tracker can be transformed into a position and orientation of the tracked object in world coordinates.

Figure 1 illustrates the basic concept. The three frames of reference (world, tracker and limb) are right hand coordinate systems. A black dot inside the circle indicates the third axis coming out of the page. An 'X' signifies the axis going into the page. The world coordinate system is fixed. For magnetic trackers it is often associated with the transmitter's antenna which does not move. The tracker's frame of reference is fixed to the

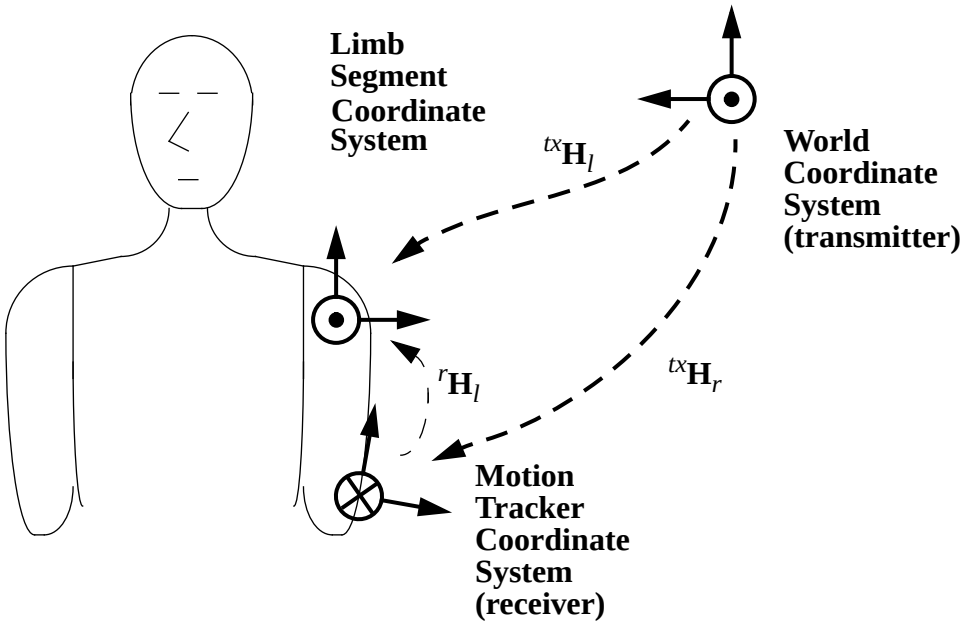

Figure 1: Relationships Between Frames of Reference

case of the magnetic receiver. The limb segment's frame of reference is fixed with respect to the underlying bone structure of the limb and is usually located at the joint whose position and angles are to be determined. If it is assumed that the tracked object is a rigid body, then the position and orientation of the limb relative to the frame of reference attached to the tracker is constant. Knowing this constant transformation matrix then makes it possible to transform tracker position and orientation in world coordinates to limb segment position and orientation in world coordinates.

If ${}^{a}\mathbf{H}_{b}$ represents a homogeneous transfer matrix for mapping coordinates given with reference to frame *b* into coordinates that are referenced to frame *a* [CRAI89], then the relationship discussed above is mathematically represented as:

$$ {}^{tx}\mathbf{H}_{l} = {}^{tx}\mathbf{H}_{r}\,{}^{r}\mathbf{H}_{l} \qquad \text{(eq 2.1)} $$

where *tx*, *r* and *l* stand for transmitter, receiver and limb segment respectively. If the limb segment is placed in a known position and orientation relative to the world coordinate system, then ${}^{tx}\mathbf{H}_{l}$ for this position will be known. While the limb segment is still in this position, data from the tracker can be read to determine ${}^{tx}\mathbf{H}_{r}$. From these two known quantities the constant ${}^{r}\mathbf{H}_{l}$ matrix can be calculated. This is basically what occurs during the calibration process. Once the world coordinates of the tracked object are known, then by knowing the dimensions of the limb segment it is possible to render a suitably scaled replication of it in the virtual environment. A more detailed analysis of calibration is provided in Chapter V.

C. PREVIOUS WORK INTERFACING HUMANS

Man-machine interfaces are necessary for the useful operation of any computer. Significant advances in the usability and application of computers have followed advances in techniques for interfacing computer users with their machines. Some examples include

development of higher level programming languages, mouse pointing devices, and graphical user interfaces (GUIs). More recently, advances in motion tracking technology have made it possible to create interfaces that permit real-time control of the highly dynamic entities that populate virtual worlds. Without these interfaces, interaction with virtual environments is greatly restricted, reducing the scope of their application.

In order to insert the human in the virtual environment, it is necessary to construct an appropriate man-machine interface. The goal is to make the interface as transparent and intuitive as possible for the user. The desire is to enhance the perception that the user is immersed in the virtual environment. At the same time, the interface must convey a large amount of information on the activities of the user to the machine. This is a tall order.

In this section, a review of some of the work on interfaces designed to insert the human in a virtual environment is presented. These implementations combine the use of computer models and motion tracking hardware as previously discussed. The focus is on systems that interface upper body movement.

1. Individual Soldier Mobility System

The *Individual Soldier Mobility System* (ISMS) is a simulator designed to allow the inclusion of dismounted infantry into large-scale simulated combat exercises. Its hardware was developed at Sarcos, Inc. Software for ISMS was developed by groups at the Naval Postgraduate School, University of Utah Center for Engineering Design, Sarcos, and University of Pennsylvania [WALD95]. Project funding was provided by the Army Research Laboratory - Human Research Engineering Directorate in late 1993. The system was first demonstrated in 1994. This demonstration marked the first time that individual dismounted soldiers could operate in a virtual environment together with traditional vehicle simulators [BADL94]. System components include the *Individual Portal* or I-PORT, through which user inputs and visual/force feedback are provided, the *Jack* human software

model, and interface software. I-PORT is currently one of the few force feedback devices available for inserting humans into the virtual environment [PRAT94, PRAT95].

The structure of ISMS as it was first demonstrated is shown in Figure 2. The I-PORT system is shown in Figure 3. To use ISMS, the soldier wears a body suit that measures upper body joint angles. He holds an instrumented rifle and sits in a room called the Walk-in Synthetic Environment (WISE) on a pedal-based mobility simulator. Computer generated imagery is provided via Head Mounted Display (HMD) and three large video screens. The suit, seat, rifle and image generator together make up I-PORT. The soldier moves through the environment by pedaling the I-PORT. He changes his direction of motion by twisting in his seat. The surface normal for the soldier's virtual position is computed and provided so that resistive feedback can be implemented consistent with terrain [GRAN95].

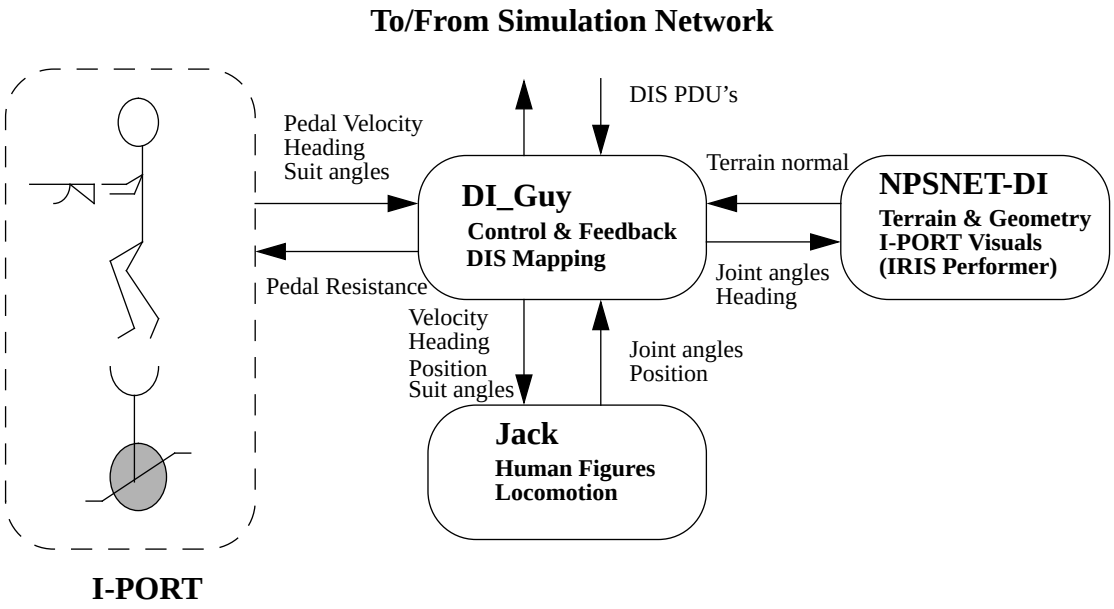

Figure 2: ISMS Structure [GRAN95]

Figure 3: IPORT Human Sensing Technology

The human model that is used as part of ISMS is called *Jack*. Jack software was developed over a two decade period at the University of Pennsylvania. Jack is a general purpose interactive environment used for manipulating articulated figures [BADL93a]. It uses its own notation (called Peabody) and was primarily designed for use as a human factors visualization tool. The human model associated with Jack is largely kinematic, though provision is made for associating strength values with joints and computing figure balance. The full Jack model includes 73 joints and 136 DOF. A general purpose constraint engine is provided with Jack that uses an iterative inverse kinematics procedure. High-level behavioral control is possible with Jack via software that manages articulation and

constraints when given a goal directed input. Jack provides the capability to simulate certain human behaviors. Not all of this functionality is needed or used in ISMS.

I-PORT and Jack are interfaced together through Naval Postgraduate School Networked Virtual Environment (NPSNET) application software. This software includes DI_Guy and NPSNET-DI. DI_Guy provides overall control of the system and information flow as shown in Figure 2. Additionally, it provides an interface with the rest of the networked virtual environment through use of Distributed Interactive Simulation (DIS) protocols. To do so it creates the appropriate protocol data units (PDU's) and sends them out over the net. NPSNET-DI manages terrain data, provides force feedback, and renders visuals and audio. The Jack model accepts heading, velocity, position and sensor suit inputs. It provides joint angles and position to the NPSNET applications for rendering. Jack provides realistic joint angles for the lower body that are in concert with inputs from the mobility simulator. These angles cause the lower body to be rendered as though the figure is standing, walking or running. Upper body angles are taken from the sensor suit, checked for validity, and passed on.

Subsequent versions of the ISMS system were redesigned to overcome significant inefficiencies in the original system [PRAT95]. In the original system, DI_Guy ran on a central server and Jack software ran on its own workstation. The new system architecture eliminated the need for this server and workstation, but created the need for a low level controller for I-PORT. Most of the functionality of the original system was converted into linkable libraries and associated with a single main NPSNET application. In the interest of efficiency, Jack was replaced by Jack Motion Library (JackML), a subset of the original software.

As can be seen from Figure 3, the upper body tracking in ISMS is accomplished through a mechanical tracker. The sensor suit of I-PORT provides accurate position and orientation in real-time, but is not without its problems. During ISMS's first demonstration

in February, 1994 some who used it felt the suit was cumbersome and difficult to adjust. The system requires a lengthy calibration process for each user. Calibration measurements prior to the demonstration were often accompanied by high noise levels that resulted in jerky motion [PRAT94]. At later demonstrations of the suit, the numbers of users made it impractical to calibrate the system for each user. As a result, it was decided that the icon lock its hands on the rifle and inverse kinematics be used to provide joint angles for the arms. While this was visually acceptable, the user could never put his rifle down in the virtual environment [PRAT94].

2. Minimally Sensed Humans and *Jack*

At the University of Pennsylvania efforts have been undertaken to track, in real-time, the posture and position of the human body using a minimal number of six DOF sensors [BADL93b]. The goal of this research is to recreate the user's upper body positioning without encumbering him with sensors. Four six DOF magnetic *Flock of Birds* trackers from Ascension Technology, Inc. are used. *Flock of Birds* is a DC-based system. Sensors are placed as shown in Figure 4, with one each on the palms, head and waist. The system is underspecified. Unsensed joint angles are determined using inverse kinematics and Jack's goal directed behavioral algorithms.

The system utilizes an Extended Range Transmitter enabling the user to move in an 8-10 foot hemisphere. Bird sensors are connected to a Silicon Graphics 310 VGX via RS232 interface operating at 38,400 baud. Special interface software is used to circumvent the IRIX operating system and allow Bird data to reach the application with minimal delay. This allows for each tracker to provide 25-30 updates per second of which only eight to ten are used. Overall performance in a shaded environment with 2000 polygons is eight to ten frames per second. This could be considered near real-time if 15 Hz is considered the minimal acceptable for real-time applications. The cause for this relatively slow

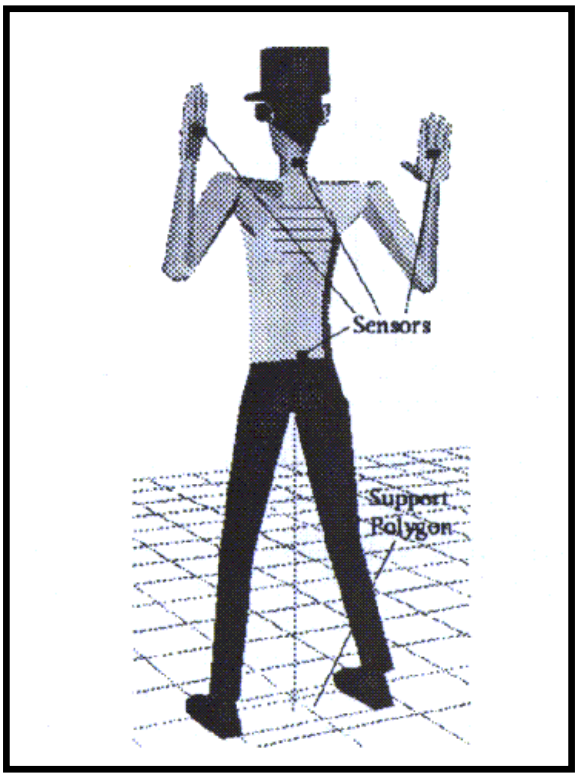

Figure 4: A Minimally Sensed Human [BADL93b]

performance is attributed directly to the inverse kinematics routine [BADL93b]. In this case, Jacobian inverse kinematics as described in [KLEI83] might work better. The routine runs in the interframe update to strike a balance between position accuracy and refresh rate. It was also noted that the system requires better noise filtering algorithms, both for the equipment and for erratic inputs from the human operator.

3. Complete and Direct Specification Systems

Recently, research at the Naval Postgraduate School has focused on improving the usability of I-PORT by replacing its mechanical sensor suit with a magnetic system. Parallel efforts were undertaken to develop systems that were completely and directly specified [MCMI96b, WALD95]. These efforts centered on modeling and tracking the human arm. In each case, an MDH model of the human arm was developed to be used with

the JackML software of ISMS. The models were designed in such a way that inverse kinematics result in joint angles that are within a constant of those utilized in Jack. This permits the interface to quickly convert results to "Jack angles" and then override existing Jack angles to render the figure in the proper posture. Both systems utilize the Polhemus *Fastrak* magnetic tracking system. The *Fastrak* system is an AC-based system with range of operation similar to *Flock of Birds*.

The completely specified system uses four six DOF trackers, one on the back of the neck, one on each wrist, and one to track a rifle. Two degrees of freedom in the wrist are fixed resulting in a 5 DOF arm. A software driver for the Polhemus sensors was written to accept data from the sensors at the highest rate available. If two sensors are utilized (i.e. one arm is tracked), then 60 Hz is the maximum available from the sensors. For all four *Fastrak* sensors, the rate is 30 Hz. Data was provided to the SGI workstation via RS232 at 9600 baud. The 9600 baud rate proved to be a bottleneck preventing sampling rates higher than 15 Hz. Faster RS232 interfaces with the SGI should be possible, but have not yet been implemented [MCMI96b]. The 15 Hz sample rate, sensor noise, and overhead associated with communicating with NPSNET and Jack, cause somewhat jerky motion to be displayed.

Work on the directly specified system remains incomplete. The system was to include four sensors, to be placed on the torso, upper arm, forearm and hand. Figure 5 shows tracker placement on the arm. With this placement, all seven DOF's of the arm could be tracked.

It is the purpose of this thesis to continue research described in [MCMI96b] and [WALD95], and extend its applicability to the entire upper body.

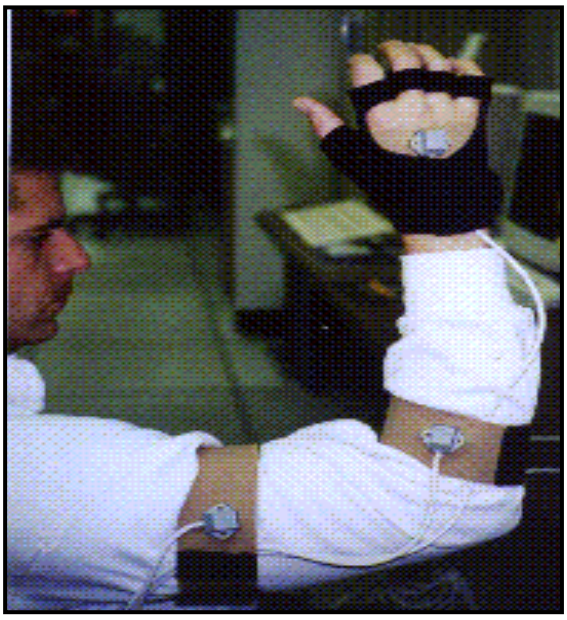

Figure 5: Sensor Position on Arm [WALD95]

D. SUMMARY

Issues concerning the modeling and tracking of humans have been presented. An overview of graphic, kinematic and dynamic modeling methods was given. Considerations for tracking humans, including motion tracker performance metrics, advantages and disadvantages of various trackers, and the placement and calibration of trackers was discussed. The chapter concluded with a survey of previous work related to this thesis. In Chapter III, forward kinematics and the model developed as part of this research is presented.

III. FORWARD KINEMATICS

The interface between the actual and virtual human makes use of both forward and inverse kinematics. Inverse kinematics is used to determine joint angles from motion tracking data. Forward kinematics is used to render the graphics (in this case in OpenGL). In this chapter, kinematics notation, modeling of the upper body, and implementation of the model in OpenGL are discussed.

A. KINEMATICS NOTATION

As mentioned, the two standardized and well known kinematic notations are the Danevit and Hartenberg (DH) and the Modified Danevit and Hartenberg (MDH) notations. The kinematic model using either of these notations specifically describes physical links (rigid bodies) connected by single DOF revolute or prismatic joints. The links are connected serially in chains or sometimes in tree-like structures. In both notations, links and joints are typically numbered in succession from the base link or root outward. Coordinate systems are assigned to each link via a standard set of rules depending on the notation used. The DH and MDH notations are equivalent, with the exception that the link frame of reference for DH links is attached to the outboard motion axis of the link, whereas the link frame of reference for MDH links is attached to the inboard motion axis. Joints in either case are indexed based upon the frame associated with them. The MDH notation is described here, though either notation can be used to model the upper body.

Figure 6 shows coordinate system assignment and the standard MDH parameters associated with links. The joint between $link_{i-1}$ and $link_i$ is labeled joint i. The coordinate frame for a link is fixed to the end of the link nearest the base. The z-axis of this frame lies along the line of motion of the joint and the x-axis along the common normal between this joint axis and the next. If the motion axes of the inboard and outboard joints intersect, then

the x-axis is assigned perpendicular to the two. If the motion axes of the joints are parallel to each other, then the x-axis lies along a common normal between them at an arbitrary location.

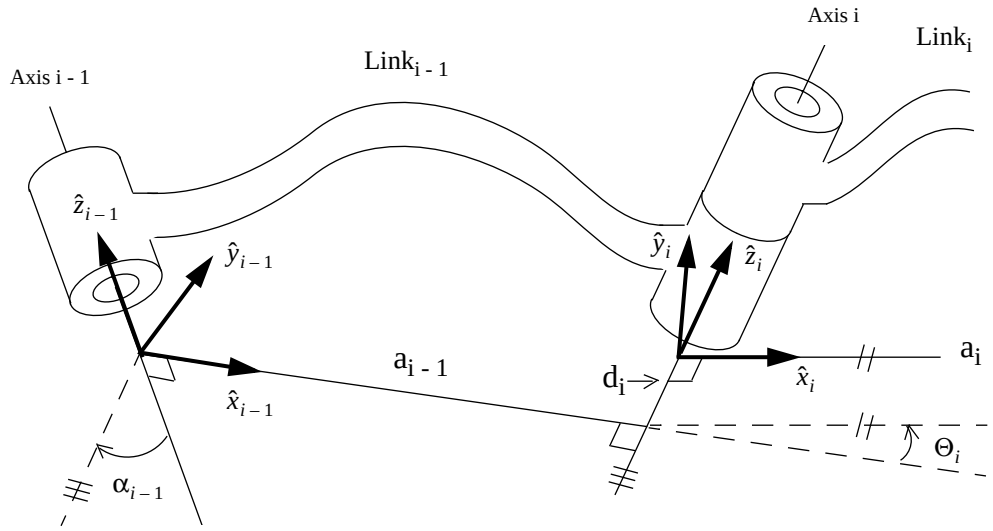

Figure 6: Link Coordinate System Assignment and MDH Parameters [CRAI89]

There are four parameters needed to describe the relationship between the frame for link$_{i-1}$ (denoted {i-1}) and the frame for link$_i$ (denoted {i}). Link$_{i-1}$ is referred to as the inboard link and link$_i$ is referred to as the outboard link. The four parameters are:

- inboard link length

  a_{i-1} = distance from $\hat{z}_{i-1}$ to $\hat{z}_i$ measured along $\hat{x}_{i-1}$

- inboard link twist

  α_{i-1} = angle between $\hat{z}_{i-1}$ and $\hat{z}_i$ measured about $\hat{x}_{i-1}$

- outboard link offset

  d_i = distance from $\hat{x}_{i-1}$ to $\hat{x}_i$ measured along $\hat{z}_i$

- outboard joint angle

  Θ_i = angle between $\hat{x}_{i-1}$ to $\hat{x}_i$ measured about $\hat{z}_i$

These parameters are constant with the following exceptions: for revolute joints Θ_i is variable, and for prismatic joints d_i is variable. The model constructed as part of this research only includes revolute joints (prismatic joints rarely occur in nature).

The transformation between {i-1} and {i} can be represented by a transformation matrix ${}^{i-1}\mathbf{T}_i$. This is found to be the result of two rotations and two translations executed as specified by the following equation:

$$ {}^{i-1}T_i = R_x(\alpha_{i-1})D_x(a_{i-1})R_z(\Theta_i)D_z(d_i) \qquad \text{(eq. 3.1)} $$

where $R_x(\alpha_{i-1})$ is a rotation about the x-axis of {i-1} of α_{i-1} degrees, and $D_x(a_{i-1})$ is a translation down the x-axis a_{i-1} units. This equation is used for rendering links in their proper position. As will later be shown, the translations and rotations are executed in the order specified by (eq. 3.1) to move from the frame of the link rendered to the frame of the next link in the series.

An equivalent expression for (eq. 3.1), found by multiplying out the transformation matrices of (eq. 3.1), is [CRAI89]:

$$ {}^{i-1}T_i = \begin{bmatrix} \cos\Theta_i & -\sin\Theta_i & 0 & a_{i-1} \\ \sin\Theta_i\cos(\alpha_{i-1}) & \cos\Theta_i\cos(\alpha_{i-1}) & -\sin(\alpha_{i-1}) & -\sin(\alpha_{i-1})d_i \\ \sin\Theta_i\sin(\alpha_{i-1}) & \cos\Theta_i\sin(\alpha_{i-1}) & \cos(\alpha_{i-1}) & \cos(\alpha_{i-1})d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(eq. 3.2)} $$

An expression for ${}^{i}\mathbf{T}_{i-1}$ can be found by calculating the inverse of (eq. 3.2) as follows [MCMI96b]:

$$ {}^{i}T_{i-1} = \begin{bmatrix} \cos\Theta_i & \sin\Theta_i\cos(\alpha_{i-1}) & \sin\Theta_i\sin(\alpha_{i-1}) & -a_{i-1}\cos\Theta_i \\ -\sin\Theta_i & \cos\Theta_i\cos(\alpha_{i-1}) & \cos\Theta_i\sin(\alpha_{i-1}) & a_{i-1}\sin\Theta_i \\ 0 & -\sin(\alpha_{i-1}) & \cos(\alpha_{i-1}) & -d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(eq. 3.3)} $$

Having described the MDH notation and the transformations that relate one link's position to the next, it is now possible to describe the upper body model that uses this notation.

B. THE KINEMATIC MODEL

The goal in creating a model of the human upper body is to provide a kinematic representation of the salient features of human upper body anatomy and motion. Since human upper body motion is defined largely by the amount of freedom of movement in the human skeleton, the model can be designed with emphasis on replicating this structure. Prominent structures include the spine, clavicles, arms and hands. The model provides only an approximation of the movement of these structures, balancing the requirements for accuracy, simplicity and speed of computation.

Figure 7 provides a sketch of one possible approximation of the human upper body skeleton. The model uses multiple DOF joints attached by rigid links. The spine is approximated by two three DOF joints placed at the waist and mid-torso. Though the actual human spine is made up of numerous two DOF vertebrae, the two three DOF joints allow the model to bend, twist, and lean in fashion similar to an actual human. Similarly, the human clavicle is a complex bone structure which has been modeled by a single two DOF joint and its associated rigid link. This allows the model to move the shoulders up and down or forward and back. The human shoulder is a ball joint and so it is modeled with a three DOF joint. The human elbow can be modeled with a single DOF rotary joint as in the actual structure. Finally, forearm roll and the two DOFs in the wrist are modeled together as a three DOF joint at the wrist. This provides for the three DOF normally seen at the hand or end-effector. Taken together, the joints provide the entire upper body model 24 DOFs.

It is now necessary to define the model mathematically using the MDH notation. It should be noted that multiple DOF joints can be modeled by chaining zero length and offset

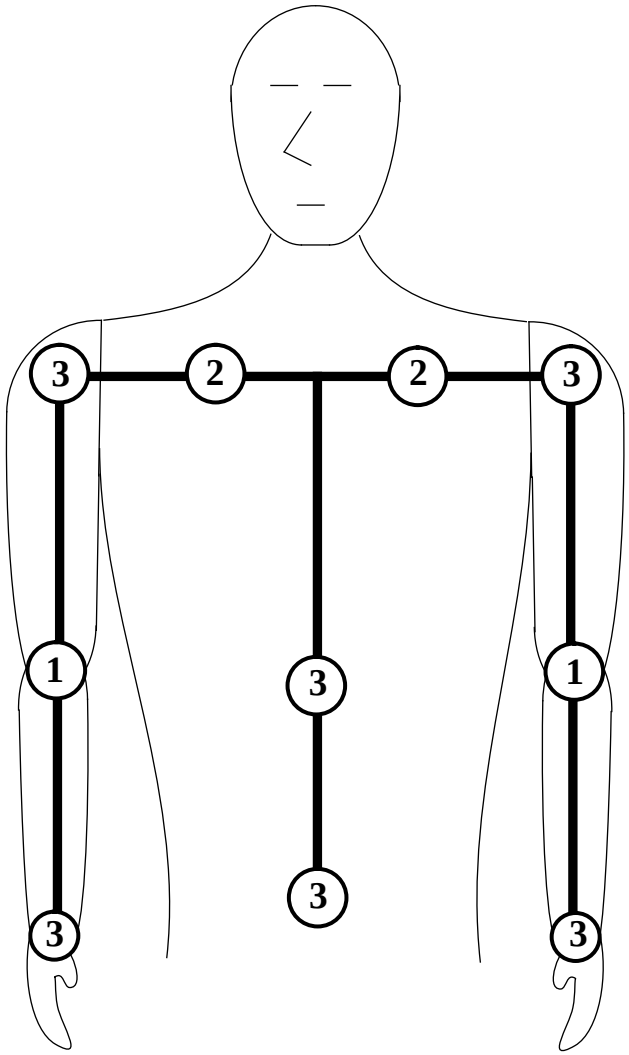

Figure 7: Upper Body Model with 24 Degrees of Freedom

links. This places the inboard and outboard axes of motion of a link at the same location. Therefore, the 24 DOF model requires 24 MDH links. The links in this case form two serial chains and are numbered starting from the base link at the waist.

Coordinate systems are assigned to each link in accordance with the rules previously defined. Figure 8 shows the result of assigning coordinate systems. The z-axes are associated with motion axes in accordance with MDH notation convention. In defining

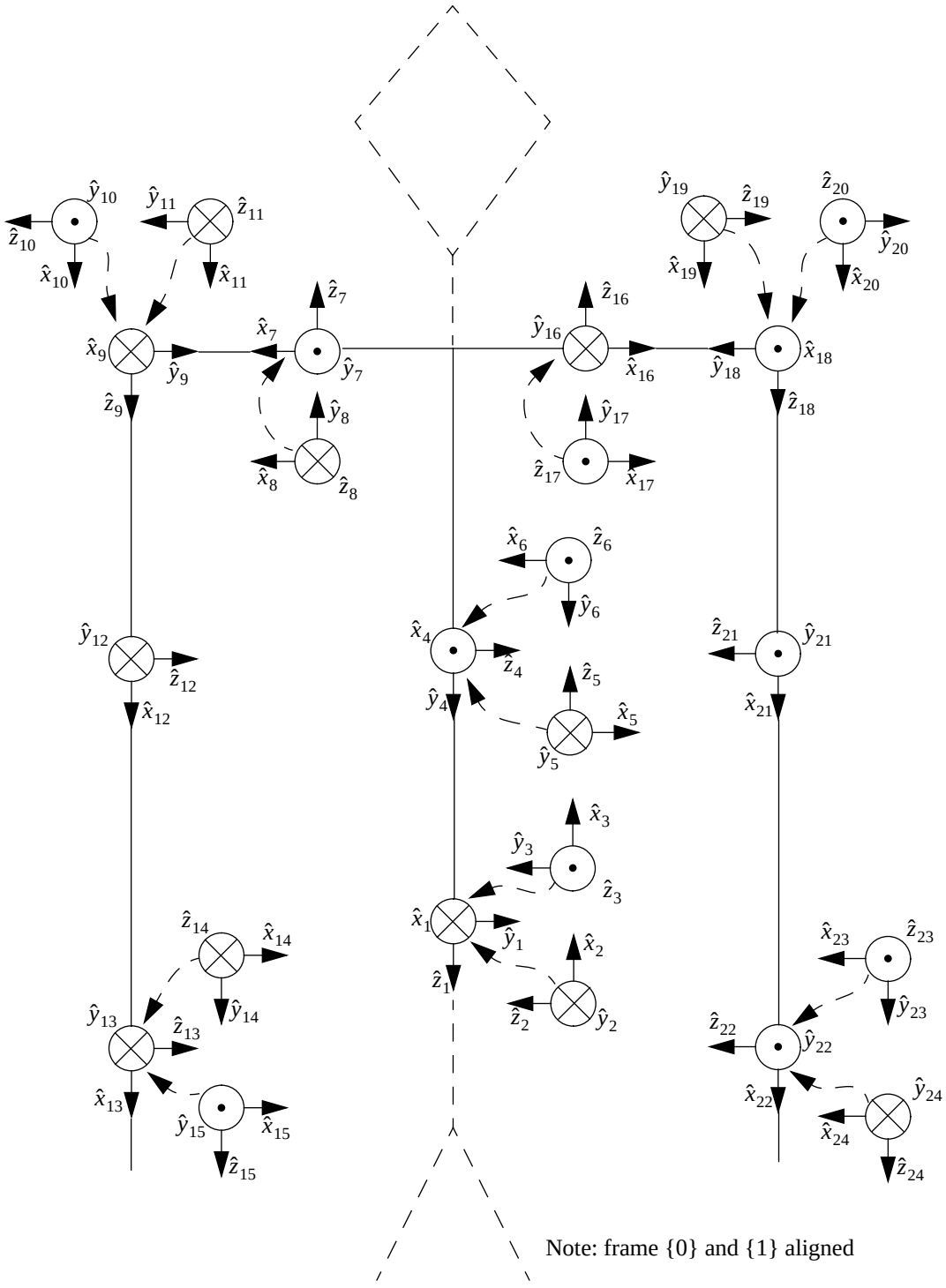

Figure 8: MDH Link Coordinate Axis

the MDH links, one must first decide the order in which motion axes are addressed. Once the order of motion axes is determined, there are two choices for each z-axis about which the motion can occur. The x-axes are also chosen in accordance with the rules previously described.

Table 3 provides a listing of MDH parameters associated with the choice of axes in Figure 8. Link lengths and offsets have been chosen somewhat arbitrarily to reflect the proportions of a typical human. These lengths can be changed later when the model is sized to the actual user of the interface. Link twists and joint angles are defined using their MDH definitions and the right hand rule. The reference joint angle for each joint, Θ_i, is that joint angle needed to position the model as shown in Figure 3. Also shown in the table are the desired limits of each joint angle. These were chosen to correspond roughly to actual limits in human motion.

Note that this is one of many possible ways to define a kinematic model with the DOFs shown in Figure 7. Different models result for a different ordering of motion axes. It is sometimes necessary to redesign a kinematic model by reordering motion axes to avoid singularities which can arise when computing inverse kinematics. This model has been shown to work well with the inverse kinematics computations which have been implemented here.

Location/ movement	**Index**	α_{i-1} (deg)	a_{i-1} (cm)	d_i (cm)	min Θ_i (deg)	ref[a] Θ_i (deg)	max Θ_i (deg)
waist twist	1	0	0	0	-90	0	90
waist bow	2	90	0	0	-270	-90	30
waist lean	3	90	0	0	-75	0	75
upper waist bow	4	90	17.5	0	0	90	180
upper waist twist	5	90	0	0	0	90	180
upper waist lean	6	90	0	0	135	180	225
left shoulder curl	7	90	7.5	22.5	-30	0	40
left shoulder shrug	8	90	0	0	-20	0	50
left shoulder roll	9	90	12.5	0	0	90	180
left arm fore-aft	10	90	0	0	0	90	270
left arm side-side	11	90	0	0	-30	0	200
left elbow curl	12	90	22.5	0	0	0	170
left hand fore-aft	13	0	25.0	0	-90	0	90
left hand side-side	14	-90	0	0	-180	-90	0
left hand roll	15	-90	0	0	-90	0	90
right shoulder curl	16	90	-7.5	22.5	140	180	210
right shoulder shrug	17	90	0	0	-20	0	50
right shoulder roll	18	90	12.5	0	0	90	180
right arm fore-aft	19	90	0	0	0	90	270
right arm side-side	20	90	0	0	-30	0	200
right elbow curl	21	90	22.5	0	-170	0	0
right hand fore-aft	22	0	25.0	0	-90	0	90
right hand side-side	23	-90	0	0	-180	-90	0
right hand roll	24	-90	0	0	-90	0	90

Table 3. MDH Kinematic Parameters

a. result in positioning as shown in Figure 8

C. IMPLEMENTATION IN C++

The kinematic model was implemented in C++ using object-oriented programming techniques. The C++ implementation of the model (forward kinematics) was used as a prototyping tool to investigate inverse kinematics. Figure 9 shows the class and object hierarchy of the software. The C++ code can be found in Appendix A.

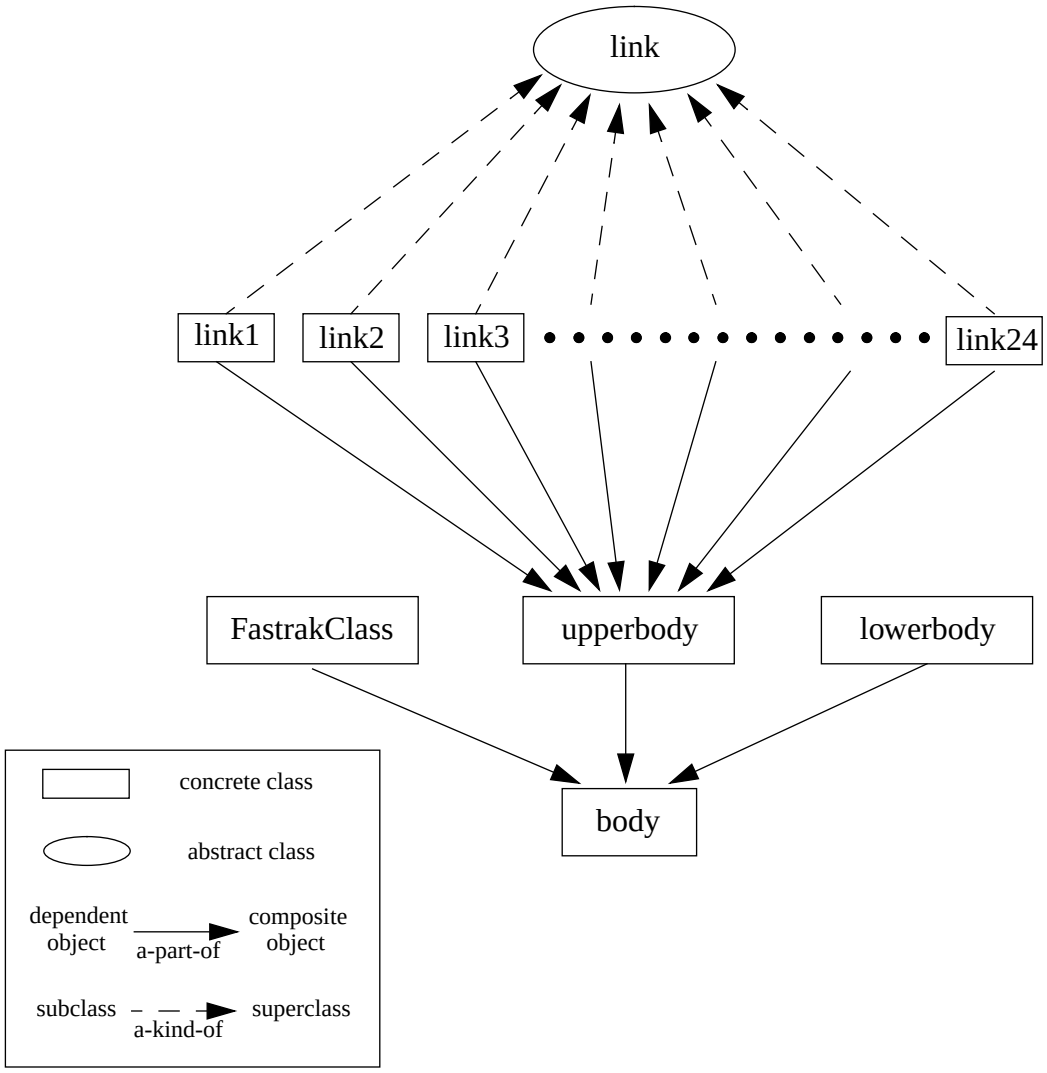

Figure 9: C++ Class and Object Hierarchy

The basic building block for the upper body structure is the link. A link object contains data members which include the four MDH parameters required to move from the previous link's frame to its own frame. This is accomplished prior to rendering the link. This is shown in the code for the **draw( )** member function of the link class in Figure 10. Forward kinematics are computed by setting a link's joint angle and then executing the code in Figure 10. Notice that this code results in the SGI computing the result of the matrix multiplications listed in (eq. 3.1). Another method to compute the forward kinematics would be to calculate the matrix found in (eq. 3.2) and then multiply the top of the OpenGL model-view stack by this matrix.

Other data members in the basic link class include the joint angle initial (or reference) position and the minimum and maximum allowable joint angles. Data members which identify parameters for drawing the link as an eight-sided diamond are also included. The 24 concrete link classes inherit these features, but are modified in two ways. First, their constructors are made to instantiate links with the parameters shown in Table 3 along with corresponding draw-parameters. Second, some link's draw functionality is modified to accommodate special requirements for that link. In most cases links draw themselves along the x-axis using a simple **drawDiamond( )** function as shown in Figure 10. In some cases this is not adequate. For example, the link6 class is responsible for drawing the head of the figure (with nose), while the link15 and link24 classes must draw the hands with thumbs (oriented initially along the z-axis). Notice that only links with positive lengths are drawn.

Links are put together in the upperbody class. The upperbody class instantiates link objects of each of the 24 link types. The upper body is drawn by drawing each of the links as shown in Figure 11. The order in which links are drawn is important. Notice that links 7 and 16 both share the same inboard link (link 6), so it is necessary to push and pop the model-view stack to draw links that form the right side of the upper body. The remaining

```
void draw()
{
   glTranslated ((GLdouble) inboard_link_length, 0.0, 0.0);
   glRotated ((GLdouble) inboard_twist_angle, 1.0, 0.0, 0.0);
   glTranslated (0.0, 0.0, (GLdouble) joint_displacement);
   glRotated ((GLdouble) joint_angle, 0.0, 0.0, 1.0);
   if (draw_length > 0.0){
      drawDiamond(0.0, 0.0, 0.0, draw_length, draw_width,
      draw_depth, draw_offset);
   }
}
```

Figure 10: Link Class Draw Member Function

functionality of the upperbody class is implemented in fashion similar to that of the **draw( )** function.

The body class object is made up of an upperbody object, a lowerbody object and a FastrakClass object. The lower body object is static, and is simply drawn as a unit in a single position and orientation. The FastrakClass object provides position and orientation data to the body so that its upper body links can be drawn in the correct position. A discussion of how this is done is found in Chapter IV. When rendered in it's reference position, the body appears as shown in Figure 12. The body is shown facing into the page with thumbs forward and elbows directly aft. It is made of 86 polygons and serves to provide visual feedback for determining the suitability of kinematics and calibration algorithms.

D. SUMMARY

The chapter began with a discussion of kinematics and the Modified Danevit-Hartenberg notation used in this research. This was followed by an explanation of the kinematic model developed. The chapter concluded with a discussion of the architecture

```
void Upperbody::draw()
{
    //draw upper body so he's facing into screne
    glRotated(90.0, 1.0, 0.0, 0.0);
    glRotated(-90.0, 0.0, 0.0, 1.0);

    link1.draw();    // draw each link, starting at the waist
    link2.draw();
    link3.draw();
    link4.draw();
    link5.draw();
    link6.draw();
    glPushMatrix();  // after drawing upper torso, remember where it was
    link7.draw();    // start drawing left side from the shoulder
    link8.draw();
    link9.draw();
    link10.draw();
    link11.draw();
    link12.draw();
    link13.draw();
    link14.draw();
    link15.draw();
    glPopMatrix();   // come back to the upper torso
    link16.draw();   // start drawing the right side from the shoulder
    link17.draw();
    link18.draw();
    link19.draw();
    link20.draw();
    link21.draw();
    link22.draw();
    link23.draw();
    link24.draw();
}
```

Figure 11: Upperbody Class Draw Member Function

of the C++ code used to graphically depict the model. In Chapter IV, Polhemus tracking of the human body is discussed. The inverse kinematics required to determine model joint angles and associated software implementations are also explained.

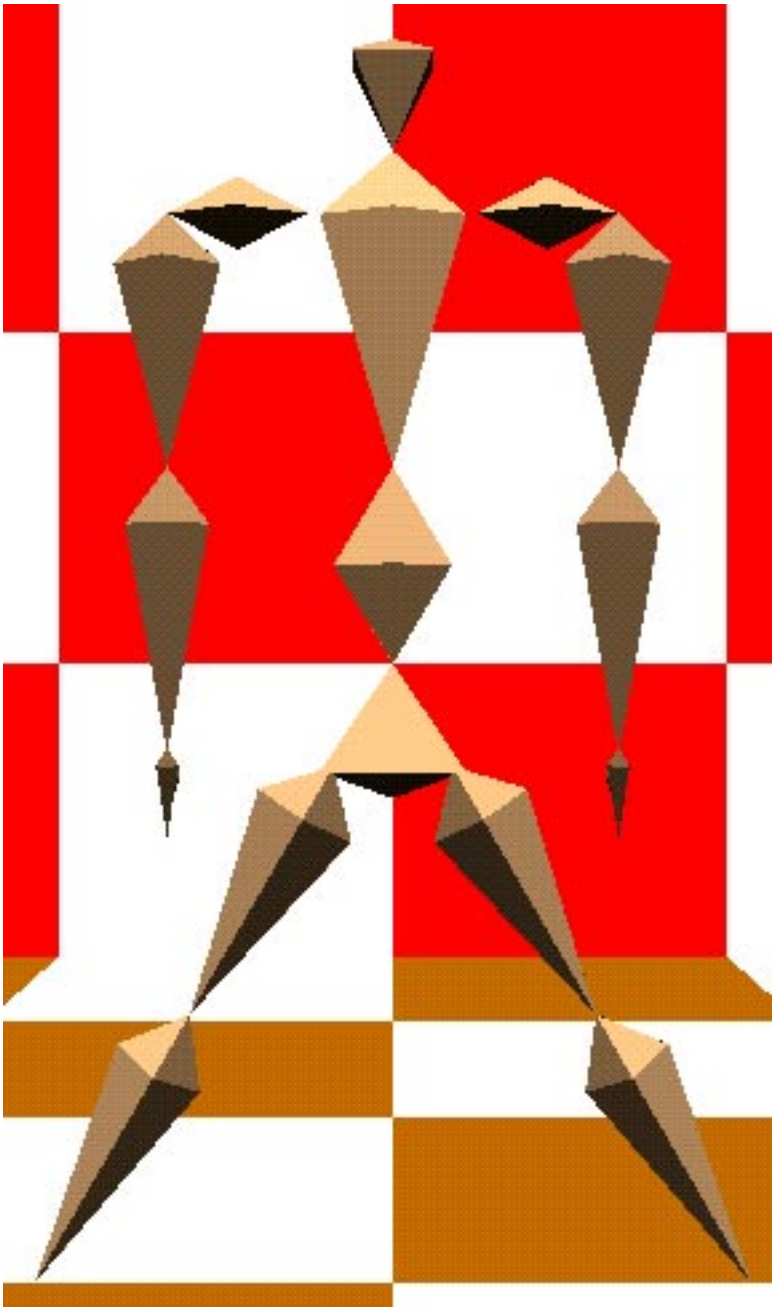

Figure 12: Body Rendered in OpenGL

IV. INVERSE KINEMATICS

Inverse kinematics is used to determine the joint angles which position the model. Polhemus *Fastrak* sensors are placed on the user's body in a manner that results in a completely specified system. Inverse kinematic computations fall into one of two distinct types; those that use angle data, and those that use position data from the trackers. In this chapter a brief overview of the Polhemus *Fastrak* system hardware and software setup is provided. This is followed by a description of tracker placement. Finally, examples of angle and position tracking inverse kinematics computations are provided.

A. POLHEMUS TRACKING

The Polhemus 3Space *Fastrak* system is an electromagnetic tracker which provides up to six DOF tracking data [POLH93]. As previously mentioned, electromagnetic position trackers work on the principle that a magnetic field induces voltage in a coil. The system includes a transmitter and up to four receivers which can be used for tracking. The emitter generates three mutually perpendicular electromagnetic fields. Each receiver contains three orthogonal coils. Each coil generates its own voltage in the presence of the electromagnetic fields for a total of nine voltages. The voltages generated are used to determine the sensor's orientation. Voltage strengths of the transmitted signals are used to determine the location of the receiver. The *Fastrak* system is an AC-based system, and as such, tends to be more susceptible to ferromagnetic interference than a DC-based system such as Ascension Technology's *Flock of Birds* [MEYE92].

1. Hardware Setup and Device Driver

The *Fastrak* hardware, device driver, and their implementation were nearly identical to that used in [MCMI96b]. A complete description of the hardware setup, device

driver, system initialization, and system operation can be found in [MCMI96b]. A brief overview along with any differences in this implementation is provided here.

Figure 13 shows the *Fastrak* hardware setup. In the center of the figure is the *Fastrak* unit. The unit is interfaced to the SGI workstation via serial cable. A DB9 connector for the computer's serial port with pins connected as shown is used to allow for software flow control. On the back of the unit are eight DIP switches used to configure the unit's serial communication. The switch setting shown specifies RS232 9600 baud connection with eight bits (no parity) and software flow control enabled. On the front of the unit, ports for the four sensors and transmitter are provided. The four DIP switches on this side of the unit are placed in the down position to enable all four sensors. Polhemus provides two types of transmitters, a standard transmitter and a "Long Ranger" which provides extended range (up to 15 ft.). In this case, the Long Ranger was used.

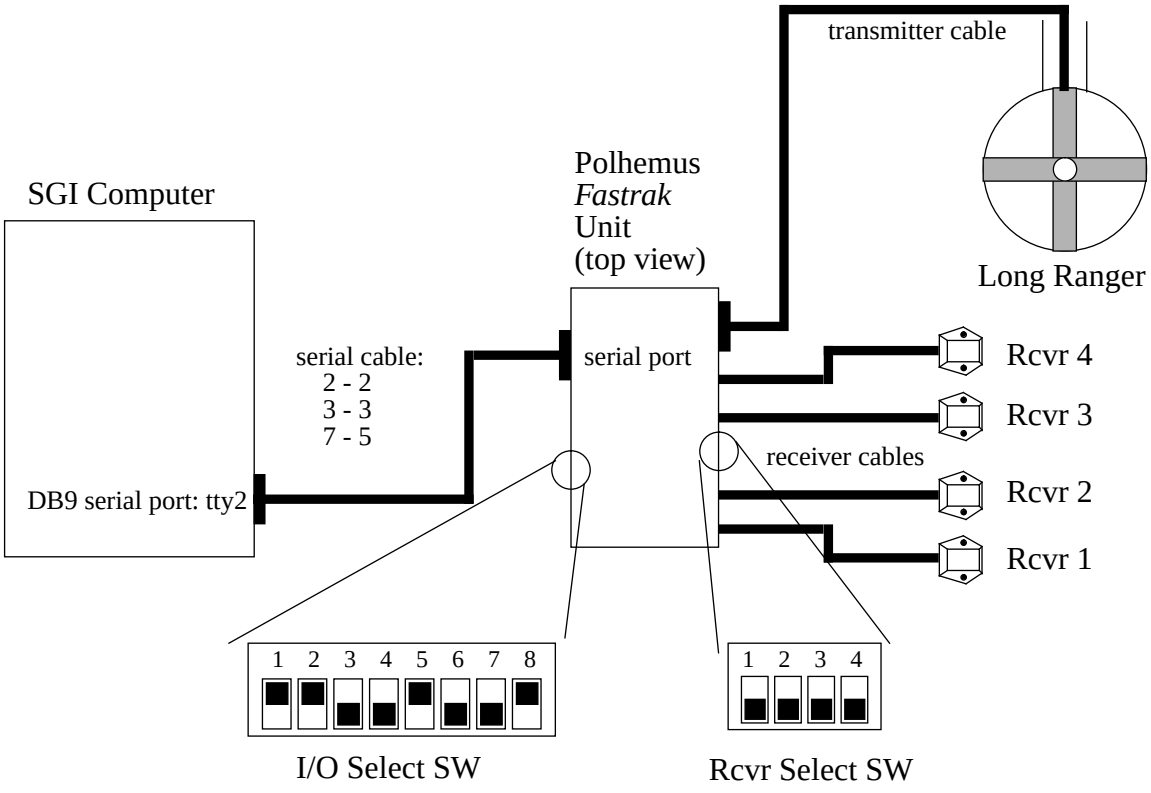

Figure 13: *Fastrak* Hardware Setup [MCMI96b]

The device driver can be found in the FastrakClass software in Appendix B. The original version of the device driver was developed for the Isotrak by Sarcos, Inc. and later revised at the Naval Postgraduate School [MCMI96b]. This version was modified by McMillan in an attempt to develop the fastest possible interface between the *Fastrak* unit and SGI workstation. McMillan found that most of the original functionality of the driver was unnecessary. The previous system polled each sensor for data and could be reconfigured during operation. Though extremely flexible in its utility, it was very slow. He therefore discarded it in favor of a faster system. The result is a driver which, 1) allows configuration only during initialization and, 2) sets up only one communication mode, a mode not previously available. The mode allows continuous binary stream of data.

Fastrak initialization occurs when the constructor of the FastrakClass is called. This happens when the Body class constructor is called. During the process, an ifstream reference to the input configuration file is passed into the FastrakClass constructor. The configuration file, called fastrak.dat, is found in Appendix C. It contains the parameters needed to specify the configuration of the *Fastrak* unit. During initialization, the constructor reads data from the configuration file, opens the I/O port, configures the *Fastrak* unit, and spawns a process which will handle the continuous stream of data. During configuration of the *Fastrak* unit, a data link between SGI and *Fastrak* unit is established. Parameters sent to the *Fastrak* unit from the SGI specify which receivers will be used along with the data types to be passed and their format. In this case, all four receivers provide homogeneous transformation matrices (orientation and position) which are passed using a 16 bit format. Parameters are also sent specifying what frame of reference will be associated with the transmitter's antennas. Figure 14 shows the frame of reference associated with receivers and transmitter. These frames are used in the inverse kinematics computations.

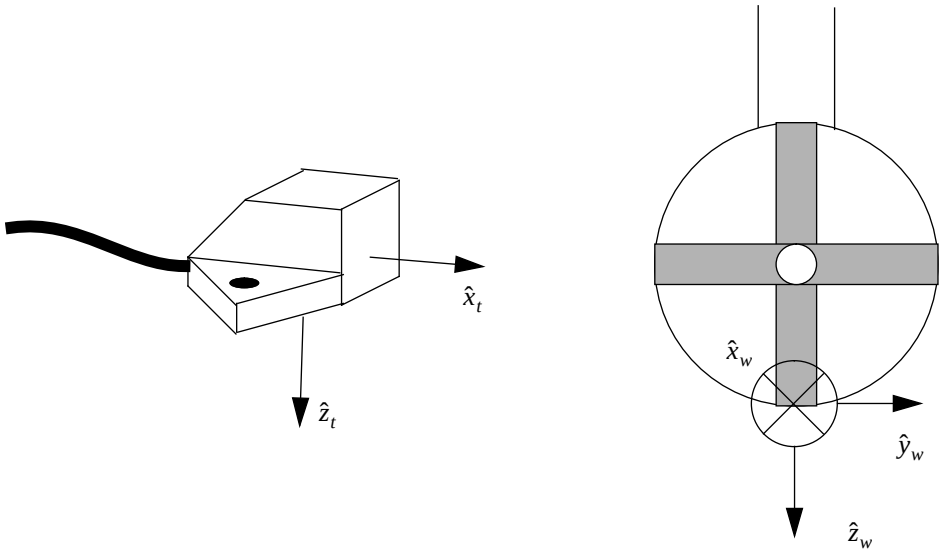

Figure 14: Coordinate System Assignment to Polhemus Devices

During continuous operation, data is read in one byte at a time. There are several actions the device driver must accomplish to make data available to the main application program. The driver must first query the port for available data. It then takes the data and processes it by first identifying the sensor to which it belongs and then converting it to IEEE floating point format. The data is then placed in buffers for access by the application program.

Figure 15 shows how data is processed during continuous operation. After the serial port is configured during initialization, a function called **pollContinuously( )** is sproc'ed. This function calls the function **getPacket( )** continuously until its parent process sends a quit signal, at which time the function is exited. The **getPacket( )** function converts the data stream into packets of data from each sensor. It does this by reading data from the serial port into a temporary character buffer **read_buffer**. It then determines whether there

is enough data to complete at least one entire packet of information from a sensor. If there is, the packets are processed and written into a second buffer **datarec_buf**. If not, then the read times out and is tried again. During the procedure, data in the stream cycles sequentially through the sensors and the **getPacket( )** process must synchronize with it. This is done by identifying a special set of header bytes which specifies a particular sensor's data. Data in **read_buffer** is discarded until **getPacket( )** synchronizes with the data stream.

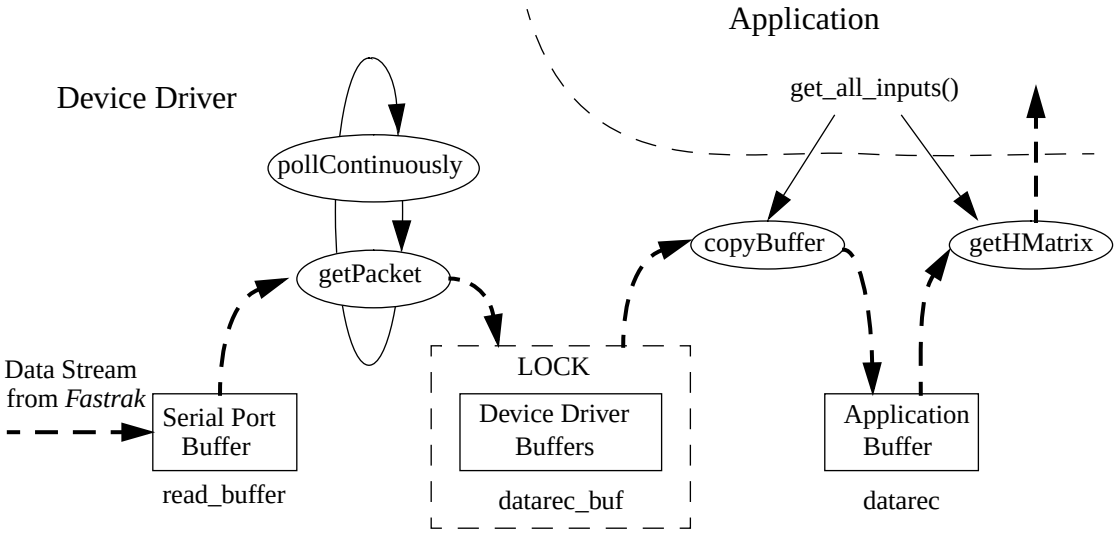

Figure 15: Software and Buffer Organization [MCMI96b]

In order to ensure that the application does not access data while it is being updated, transfer of data to the second buffer is protected by a lock and mutual exclusion. To minimize access to **datarec_buf**, the Body class function **get_all_inputs( )** calls the FastrakClass function **copyBuffer( )** to copy the entire contents of **datarec_buf** into a third buffer **datarec**. The **get_all_inputs**( ) function then calls the FastrakClass function

getHMatrix( ) four times to acquire data for each sensor and place it into Body class data members. The data is then used by the Body class object to determine joint angles.

The use of the Body class **get_all_inputs( )** function is the only difference between this application of the device driver and the one used by McMillan [MCMI96b]. In McMillan's application of the device driver a separate **get_multival( )** application function was used to call **getHMatrix( )**. In both applications, **get_all_inputs( )** is called at the beginning of each frame.

2. Tracker Placement

Sensor placement is determined by the requirement to have a completely specified system. Use of six instead of three DOF motion trackers results in a less encumbered user. The six DOF data from these trackers (position and orientation) can be used to determine up to six joint angles without ambiguity if the position of the base or first joint is known [CRAI89]. In the case of three DOF motion trackers (orientation-only trackers), only three angles can be determined. This results in a requirement for a tracker to be placed on every limb segment, or up to twice as many trackers.

a. Optimal Tracker Placement

Figure 16 shows optimal positioning of the trackers for the model chosen. The placement is optimal because it results in the fewest number of trackers being used to determine all 27 DOF inherent in the model (24 joint angles plus the base position in x, y, and z coordinates). Notice that six trackers are required, whereas only four are available with the Polhemus system used. Notice also that trackers are placed on every other physical limb, starting from the hands inward towards the base link. This is due to the fact that the hands (end-effectors) must be tracked to determine their orientation. This cannot be done by placing a tracker on the forearm. The forearm joint angles, however, can be determined from data from trackers on the hand and upper arm. The upper arm tracker

provides information on the position of the elbow joint. This information together with position and orientation information from the hand sensor is adequate to determine the elbow angle. If the upper arm sensor is removed and placed on the clavicle to provide position information on the shoulder, then information from the hand sensor is required to determine seven joint angles between it and the shoulder. The result is an infinite number of possible solutions since the system is redundant. A similar logic is used to determine remaining placement of trackers on the upper torso and on the pelvis just below $link_1$. The pelvis, or base link, is considered $link_0$. The tracker on the base link is the reference tracker for providing lower body position and orientation. All joint angles are ultimately determined relative to this base (frame {0}) position and orientation.

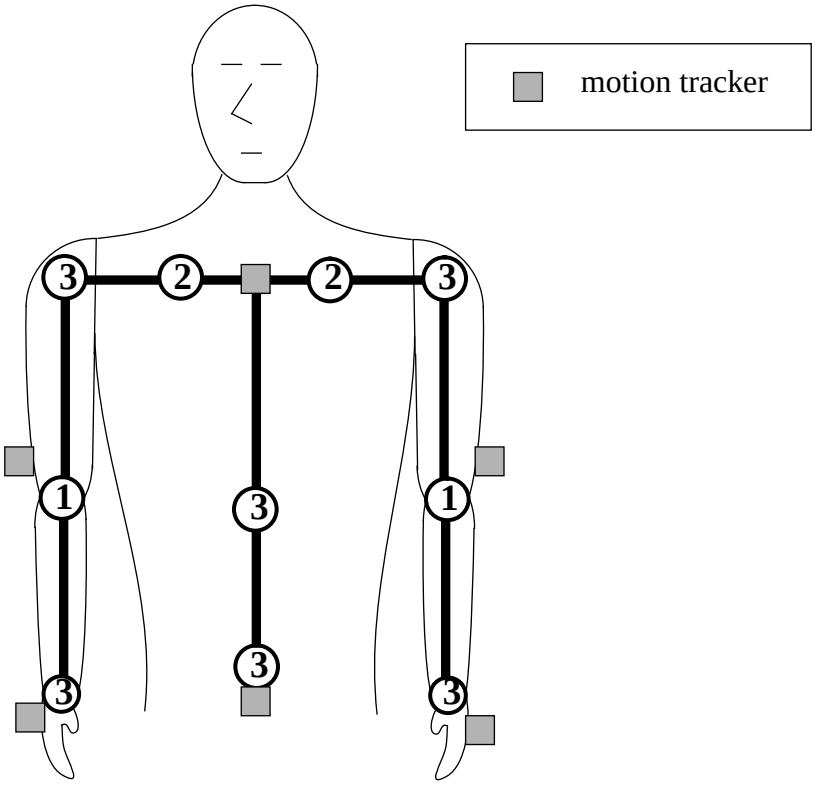

Figure 16: Optimal Motion Tracker Placement

b. Actual Tracker Placement

Figure 17 shows the actual tracker placement used in this research. This tracker placement was chosen to investigate angles-only tracking techniques involving the lower waist and right shoulder, elbow and wrist. Following the angles-only investigation, an investigation of position tracking of the right clavicle was conducted using the torso and upper arm trackers.

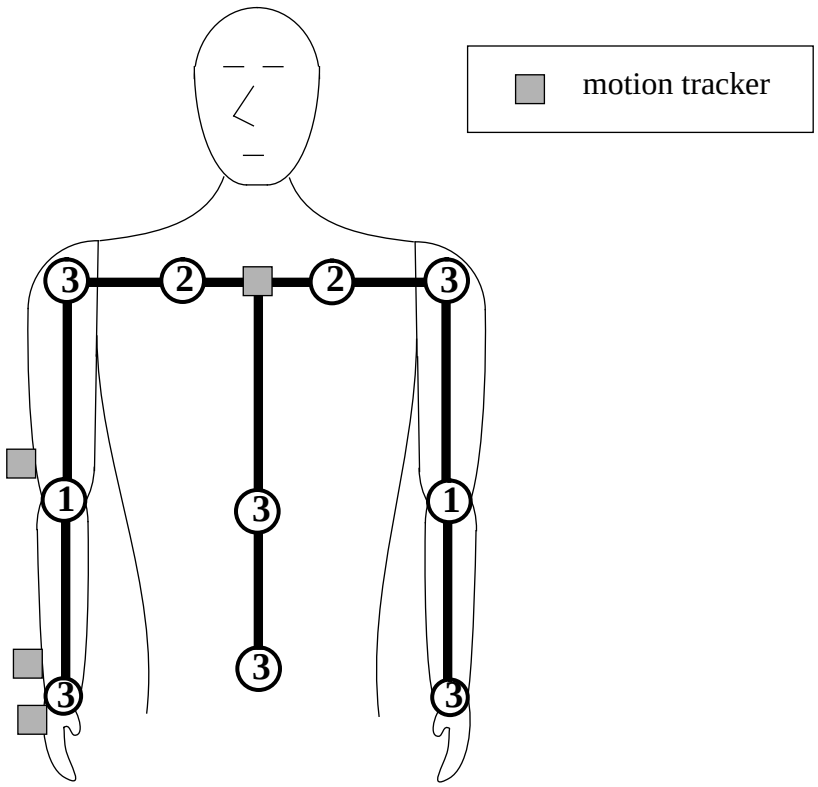

Figure 17: Actual Motion Tracker Placement

Tracker locations and methods of attachment provide for two key considerations. First, tracker position is reasonably stable relative to the underlying bone structure throughout the range of motion. Second, the equipment is easy to don. The upper arm and lower arm sensors were placed near the elbow and on the wrist by means of elastic

velcro straps placed either directly on the arm or tightly over clothing. These locations provide the least amount of relative motion between tracker and underlying bone structure due to the muscles of the arm. The hand sensor was sewn to the back of a glove worn on the hand. The torso sensor was originally worn on the back using an elastic harness comprised of two loops through which the user placed his arms, the method used in [MCMI96b]. This placed the torso tracker directly between the shoulder blades and approximately in line with the two shoulder joints. It was found, however, that movement of the clavicle of either shoulder caused considerable movement of this base tracker relative to the back. This resulted in skewing joint angle computations. Movement was due to the close proximity and motion of the shoulder blades. For this reason, an alternative method for attaching the torso sensor was investigated. Figures 18 and 19 show sensor attachment and the new harness. A discussion of the effectiveness of the new harness can be found in Chapter VI.

Figure 18: Polhemus *Fastrak* Sensor Attachment - Arm Sensors

3. Transforming Tracker Data

As discussed in Chapter II, the first step in using Polhemus tracker data to determine joint angles concerns transforming data on the tracker's position and orientation into data

a) Front b) Back

Figure 19: Polhemus *Fastrak* Sensor Attachment - Upper Body Harness

on the tracked limb segment's position and orientation. Figure 20 demonstrates how this is done. If the limb segment is treated as a rigid body (ie. tracker placement is such that the underlying bone structure moves very little in relation to the tracker) then the relationship between the tracker's frame of reference and the limb's frame of reference is constant and represented by the following:

$$^{w}\mathbf{H}_{l} = {}^{w}\mathbf{H}_{t}{}^{t}\mathbf{H}_{l} \qquad \text{(eq. 4.1)}$$

In this case, the goal is to determine the homogeneous transformation matrix representing the limb's position and orientation in world coordinates, $^{w}\mathbf{H}_{l}$. Recall that the world coordinate system is attached to the transmitter's antennas (Figure 16). $^{w}\mathbf{H}_{t}$ is the homogeneous transformation matrix reported by the tracker relative to the world coordinate system. $^{t}\mathbf{H}_{l}$ is the homogeneous transformation matrix specifying the limb segment frame relative to the tracker's frame and is constant. This can be found using a calibration

process. A more complete description of this process is contained in Chapter V. ${}^{t}\mathbf{H}_{l}$ can then be used to transform tracker data into limb segment data using (eq. 4-1).

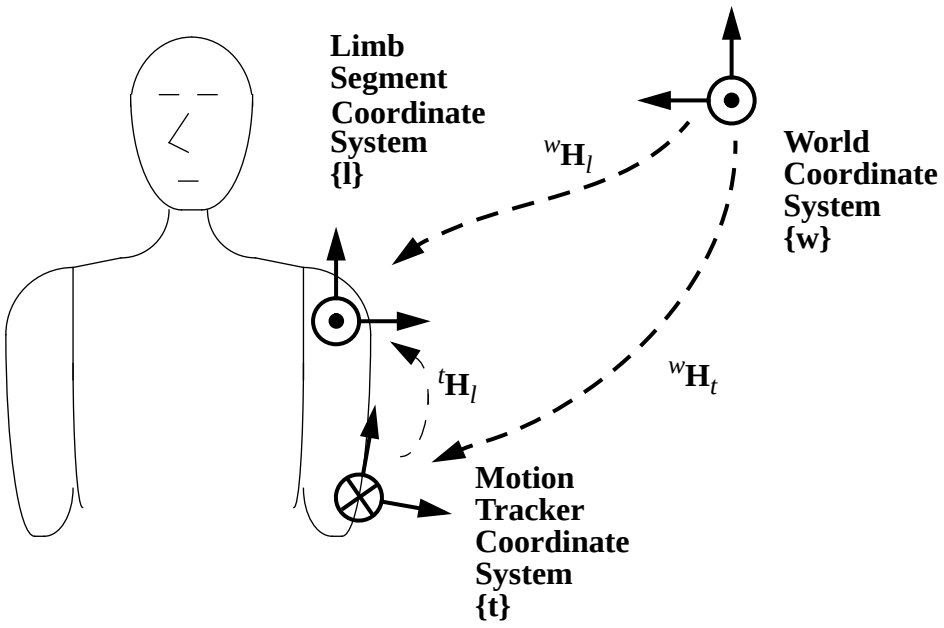

Figure 20: Relationships Between Frames of Reference

B. INVERSE KINEMATICS

The remaining inverse kinematics computations for determining joint angles fall into two categories, those that use angle or orientation information from the trackers, and those that use position information from the trackers. In general, orientation data is used to determine joint angles associated with the physical limb on which the tracker is mounted. Position information is used to determine joint angles associated with physical limbs which do not have trackers mounted on them. An example of each type of calculation is provided here.

1. Using Angle Data

The first example shows how joint angles can be determined from tracker orientation data. In this example, tracker placement is assumed to be optimal (Figure 16). The actual implementation found in Appendix A uses a similar technique and is explained later in this chapter.

The goal is to determine the three joint angles for the shoulder using orientation data from the trackers. For a tracker placed on the right upper arm, the tracker data can be transformed into a homogeneous transformation matrix defining $link_{20}$ position and orientation in world coordinates, or $\mathbf{H}_{20}$. Additionally, $\mathbf{H}_{20}$ is represented by the following:

$$\mathbf{H}_{20} = \mathbf{H}_{body}\ {}^{0}\mathbf{T}_{1}{}^{1}\mathbf{T}_{2}{}^{2}\mathbf{T}_{3}{}^{3}\mathbf{T}_{4}{}^{4}\mathbf{T}_{5}{}^{5}\mathbf{T}_{6}{}^{6}\mathbf{T}_{16}{}^{16}\mathbf{T}_{17}{}^{17}\mathbf{T}_{18}{}^{18}\mathbf{T}_{19}{}^{19}\mathbf{T}_{20} \qquad \text{(eq. 4.2)},$$

where $\mathbf{H}_{body}$ is provided by a motion tracker on the base link, or $link_0$. The T- matrices are determined by using (eq. 3.2) and the parameters for each link found in Table 3. If the joint angles for links in the chain up to the three links associated with the tracked limb have been previously determined from intermediate tracker data, then

$$ {}^{0}\mathbf{T}_{17} = {}^{0}\mathbf{T}_{1}{}^{1}\mathbf{T}_{2}{}^{2}\mathbf{T}_{3}{}^{3}\mathbf{T}_{4}{}^{4}\mathbf{T}_{5}{}^{5}\mathbf{T}_{6}{}^{6}\mathbf{T}_{16}{}^{16}\mathbf{T}_{17} \qquad \text{(eq. 4.3)}$$

is known. Rearranging (eq. 4.2) to place the known quantities on the left side of the equation,

$$ {}^{17}\mathbf{T}_{0}\,(\mathbf{H}_{body})^{-1}\,\mathbf{H}_{20} = {}^{17}\mathbf{T}_{18}{}^{18}\mathbf{T}_{19}{}^{19}\mathbf{T}_{20} = {}^{17}\mathbf{T}_{20} \qquad \text{(eq. 4.4)}$$

where ${}^{17}\mathbf{T}_{0}$ is the inverse of ${}^{0}\mathbf{T}_{17}$. Using the tracker data and known joint angles, a 4 x 4 matrix of data representing ${}^{17}\mathbf{T}_{20}$ can be calculated. Again, by using (eq. 3.2) it can be shown that by multiplying out the T-matrices ${}^{17}\mathbf{T}_{18}$, ${}^{18}\mathbf{T}_{19}$, and ${}^{19}\mathbf{T}_{20}$, that ${}^{17}\mathbf{T}_{20}$ is:

$$ {}^{17}T_{20} = \begin{bmatrix} c_{18}c_{19}c_{20} + s_{18}s_{20} & -c_{18}c_{19}s_{20} + s_{18}c_{20} & c_{18}s_{19} & 0 \\ -s_{19}c_{20} & s_{19}s_{20} & c_{19} & 0 \\ s_{18}c_{19}c_{20} - c_{18}s_{20} & -s_{18}c_{19}s_{20} - c_{18}c_{20} & s_{18}s_{19} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(eq. 4.5)} $$

where notations such as $s_{18} = \sin \Theta_{18}$ and $c_{18} = \cos \Theta_{18}$ are used. The task now becomes one of determining the joint angles Θ_{18}, Θ_{19} and Θ_{20} from this matrix. If one lets

$$ {}^{17}\mathbf{T}_0 \, (\mathbf{H}_{body})^{-1} \, \mathbf{H}_{20} = \begin{bmatrix} A_1 & B_1 & C_1 & D_1 \\ A_2 & B_2 & C_2 & D_2 \\ A_3 & B_3 & C_3 & D_3 \\ A_4 & B_4 & C_4 & D_4 \end{bmatrix} \qquad \text{(eq. 4.6)} $$

then,

$$ \sin \Theta_{19} = \pm \sqrt{A_2^2 + B_2^2} \qquad \text{(eq. 4.7)} $$

and

$$ \sin \Theta_{19} = \pm \sqrt{C_1^2 + C_3^2} \qquad \text{(eq. 4.8).} $$

Since normally $0 < \Theta_{19} < 180$ degrees, $\sin \Theta_{19}$ is chosen to be positive. The three angles associated with this solution can now be found as follows:

$$ \Theta_{19} = \text{atan2} \, (\sin\Theta_{19}, C_2) \qquad \text{(eq. 4.9)} $$

$$ \Theta_{20} = \text{atan2} \, (B_2/\sin\Theta_{19}, -A_2/\sin\Theta_{19}) \qquad \text{(eq. 4.10)} $$

$$ \Theta_{18} = \text{atan2} \, (C_3/\sin\Theta_{19}, C_1/\sin\Theta_{19}) \qquad \text{(eq. 4.11)} $$

There are two problems which may be encountered with this general type of solution. The first concerns the fact that $\sin\Theta_{19}$ may equal zero. In this case, a singularity has been encountered and joint_{18} and joint_{20} axes of rotation are in line with one another.

Care must be taken so that the model is designed such that for the starting or reference position (Figure 8) this does not occur. However, the singularity may be entered after tracking commences. There are two possibilities for handling this. The first is avoidance. When $\sin\Theta_{19}$ nears zero simply set it to some arbitrarily small value. If, however, the application demands that the singularity be represented more accurately, the method described in [MCMI96b] can be used. In this case, set either Θ_{18} or Θ_{20} to its previously known value and then solve for the other angle. This can be done by noting that $\cos\Theta_{19} = 1$. Then

$$A_1 = c_{18}c_{20} + s_{18}s_{20} = \mathbf{cos}(\Theta_{18} - \Theta_{20}) \qquad \text{(eq. 4-12)}$$

$$A_3 = s_{18}c_{20} - c_{18}s_{20} = \mathbf{sin}(\Theta_{18} - \Theta_{20}) \qquad \text{(eq. 4-13)}$$

and

$$\Theta_{18} - \Theta_{20} = \mathbf{atan}\frac{A_3}{A_1} \qquad \text{(eq. 4-14).}$$

Perhaps the best method of handling singularities, a method not used here, is to design the model so that singularities are never possible. This can be done for three DOF joints by reordering the axes of rotation to ensure that the second axis of rotation can never be rotated to align the first and third axes of rotation. For example, if the first and second axes of rotation in the model used here (lower body twist and fore-aft bending at the waist) are switched in order, then no singularity between the first and third motion axes is possible. This is due to the fact that humans cannot twist the full 90 degrees required to align the first and third axes.

The second problem concerns the case where Θ_{19} is not limited to an angle between 0 and 180 degrees. In this case $\sin\Theta_{19}$ can become a negative number. A method for

determining when the angle has moved beyond this range of motion is required. If Θ_{18} is constrained to being between 0 and 180 degrees, then the sign of C_3 is the same as the sign of $\sin\Theta_{19}$. In such a case, a negative value for C_3 indicates that Θ_{19} is no longer between 0 and 180 degrees. The result of (eq. 4.7) or (eq. 4.8) is therefore chosen to be negative.

2. Using Position Data

To demonstrate how position data can be used to determine joint angles, the method for determining the clavicle joint angles Θ_{16} and Θ_{17} will be described. Figure 21 will be used to show how this can be done. The goal is to determine the position vector between the clavicle and shoulder joints. This can be done from position and orientation data given by the trackers on the back and upper arm. This vector is then used to determine the necessary joint angles.

The first problem is to determine ${}^{w}p_{16to18}$, the free vector in world coordinates describing the relative positions of joint$_{16}$ and joint$_{18}$. Figure 22 shows the vectors needed to accomplish this. Once again, one makes the assumption that the trackers are attached to links that are rigid bodies, and that the relative motion between tracker and link is nil. With this assumption, ${}^{ts}p_{posit16}$, the position of joint$_{16}$ relative to the torso sensor or tracker, and ${}^{uas}p_{posit18}$, the position of joint$_{18}$ relative to the upper arm sensor, are constants which can be determined through calibration. If ${}^{w}p_{ts}$ and ${}^{w}p_{uas}$ position vectors are provided by the trackers, then

$$ {}^{w}p_{joint16} = {}^{w}p_{ts} + {}^{w}\mathbf{R}_{ts}\,{}^{ts}p_{posit16} \qquad \text{(eq. 4.15)} $$

and

$$ {}^{w}p_{joint18} = {}^{w}p_{uas} + {}^{w}\mathbf{R}_{uas}\,{}^{uas}p_{posit18} \qquad \text{(eq. 4.16),} $$

where ${}^{w}\mathbf{R}_{ts}$ and ${}^{w}\mathbf{R}_{uas}$ are rotation matrices provided by the trackers. Once ${}^{w}p_{joint16}$ and ${}^{w}p_{joint18}$ are found, ${}^{w}p_{16to18}$ can be determined from the following relationship:

$$ {}^{w}p_{16to18} = {}^{w}p_{joint18} - {}^{w}p_{joint16} \qquad \text{(eq. 4.17).} $$

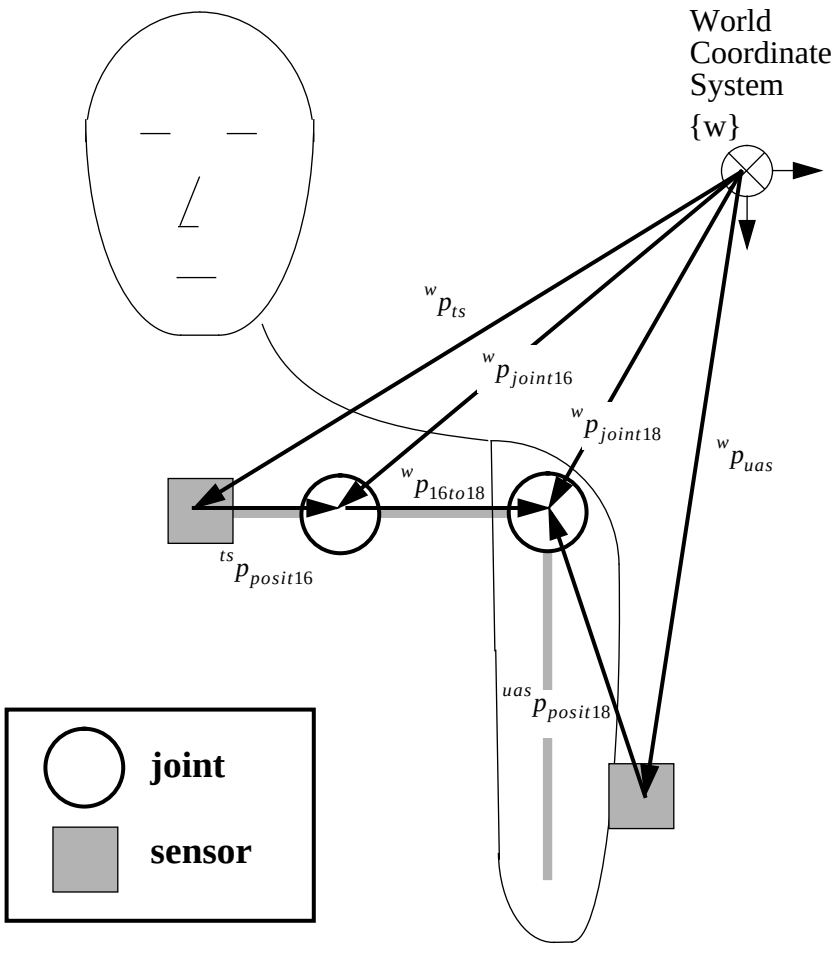

Figure 21: Clavicle Position Tracking

Equivalently, the position vectors ${}^{w}p_{joint16}$ and ${}^{w}p_{joint18}$ can also be calculated using homogeneous transformation matrices. If ${}^{w}\mathbf{H}_{ts}$ and ${}^{w}\mathbf{H}_{uas}$ are the homogeneous transformation matrices provided by the trackers, then calibration can provide the constant matrices ${}^{ts}\mathbf{H}_{posit16}$ and ${}^{uas}\mathbf{H}_{posit18}$ which can be used to transform positions reported at the trackers to the positions of $joint_{16}$ and $joint_{18}$ in world coordinates. This is accomplished using the following relationships:

$${}^{w}\mathbf{H}_{posit16} = {}^{w}\mathbf{H}_{ts}\ {}^{ts}\mathbf{H}_{posit16} \qquad \text{(eq. 4.18)}$$

$${}^{w}\mathbf{H}_{posit18} = {}^{w}\mathbf{H}_{uas}\ {}^{uas}\mathbf{H}_{posit18} \qquad \text{(eq. 4.19)}$$

In this case, the position vectors and rotation matrices previously described can be found in the homogeneous transformation matrix with the same sub/superscripts.

Now that ${}^{w}p_{16to18}$ has been determined, it is necessary to describe this vector in coordinates relative to a coordinate system attached to the upper part of the torso. This is necessary since the angles that need to be determined, Θ_{16} and Θ_{17}, are used to position the clavicle relative to the torso. The homogeneous transformation matrix $\mathbf{H}_6$ is the position and orientation of the upper torso (specifically $link_6$) in world coordinates. This has been previously determined from the tracker on the upper torso. The rotation matrix $\mathbf{R}_6$, the orientation of the upper torso given in $\mathbf{H}_6$, is needed to transform ${}^{w}p_{16to18}$ into a vector relative to the upper torso. This can be done using the following equation:

$${}^{6}p_{16to18} = (\mathbf{R}_6)^{-1}\ {}^{w}p_{16to18} \qquad \text{(eq. 4.20).}$$

The joint angles can now be determined directly from the coordinates given in ${}^{6}p_{16to18}$ using trigonometry. If *x*, *y*, and *z* are the coordinates of ${}^{6}p_{16to18}$, then

$$\Theta_{16} = \text{atan2}\,(z, x) \qquad \text{(eq. 4.21)}$$

and

$$\Theta_{17} = \text{atan2}\ (-y, \sqrt{x^2 + z^2}) \qquad \text{(eq. 4.22).}$$

The result of (eq. 4.22) must be normalized to a positive angle for use with the kinematic model.

It should be noted that the position tracking technique shown here is only valid for determining up to two joint angles. If the physical limb being tracked can change its pitch, azimuth and roll, then there is no way of determining the roll from a position vector generated by this technique. This is due to the fact that the position vector generated is parallel to the axis of roll. Only limb azimuth and pitch can be determined. In the case of a three DOF limb, it is still possible to determine all three angles. Pieper investigated inverse kinematic solutions for six DOF manipulators where three consecutive motion axes intersect [PIEP68, PIEP69]. His solution for manipulators where the last three motion axes intersect is appropriate for determining Θ_1, Θ_2 and Θ_3, given the optimal tracker placement shown in Figure 16. A description of this technique can be found in [CRAI89].

3. Implementation Specifics

There are two implementations which were investigated. Each uses the techniques shown in the previous two sections to determine joint angles or track positions.

a. Angles-only Tracking Implementation

In the angles-only tracking implementation, the objective was to demonstrate that four sensors could be used to fully articulate one arm while at the same time tracking the torso position and orientation. Though not optimal, this provides a usable solution, for the user is able to move around the virtual environment and use one arm. Since only tracker orientation data is used, many of the joint angles of the upper body model are not articulated. Table 4 shows the joint angles that are computed. The code for calculating

the joint angles can be found in Appendix A in the **calculate_joint_angles( )** function of the Body class. In all, 13 DOF are tracked with this solution.

A major difference between this implementation and the optimal implementation concerns the placement of the base tracker. The base tracker is positioned on the upper torso as in Figure 17, instead of on the pelvis. Since there is no tracking of the lower body, the lower body is not rendered or displayed. The three joints located at the upper-waist position, Θ_4, Θ_5 and Θ_6, are fixed and allow for the entire back to be treated as a rigid body. Since this is the case, the position tracking techniques described in section 2 above can be used to determine the base position of the upper body from the position and orientation data of the torso sensor. The upper body is translated to this position before it is rendered. Angle tracking techniques from section 1 above are then used to determine joint angles given in Table 4, the only difference being that joint angles which are fixed (ie. not tracked) result in constant T-matrices.

Joint Angle	Index	Trackers	Type Computation
waist twist	1	torso	angle
waist bow	2	torso	angle
waist lean	3	torso	angle
right shoulder roll	18	upper arm	angle
right arm fore-aft	19	upper arm	angle
right arm side-side	20	upper arm	angle
right elbow curl	21	lower arm	angle
right hand fore-aft	22	hand	angle
right hand side-side	23	hand	angle
right hand roll	24	hand	angle

Table 4. Joint Angles Computed - Angles-only Implementation

b. Position Tracking Implementation

In the position tracking implementation, the objective was to demonstrate position tracking of the right clavicle. The code for calculating the joint angles can be found in Appendix D in the revised **calculate_joint_angles( )** function of the Body class. Table 5 shows the joint angles that are calculated. The computations used to determine Θ_{16} and Θ_{17}, are exactly those shown in section 2 above.

Joint Angle	Index	Trackers	Type Computation
waist twist	1	torso	angle
waist bow	2	torso	angle
waist lean	3	torso	angle
right shoulder curl	16	torso & upper arm	position
right shoulder shrug	17	torso & upper arm	position

Table 5. Joint Angles Computed - Position Implementation

C. SUMMARY

In this chapter the Polhemus tracking hardware and software were discussed. The inverse kinematics computations were also presented. It was hoped that success in tracking the clavicle would result in an attempt to use position tracking for determining the joint angle for the lower arm. This would result in progress towards the optimal tracker placement shown in Figure 16. This was not the case, however, due to limitations in tracker hardware. A description of results is contained in Chapter VI. The next chapter, Chapter V, discusses methods used to calibrate the system.

V. CALIBRATION

In this chapter a discussion of the methods used to calibrate the sensors and size the kinematic model to the user is provided. The goal of this effort is to make the calibration process simple to use, yet sufficiently accurate to permit the user to effectively interact with his virtual environment.

A. CALIBRATING SENSORS

The first step in using Polhemus tracker data to determine joint angles is to transform data on the tracker's position and orientation into data on the tracked limb segment's position and orientation. If the tracked limb segment is treated as a rigid body, then it can be assumed that movement of the tracker relative to the limb segment is nil. This assumption is only valid if, 1) the limb segment's frame is attached to the underlying bone structure of the limb and, 2) the tracker is attached to the limb in a manner that minimizes its motion relative to this same bone structure. This can be done by attaching the tracker at a location where there is little muscle or flesh matter between the skin and underlying bone. Once the tracker is properly placed, it is the purpose of calibration to determine the constant 4 x 4 homogeneous transformation matrix which describes the relationship between tracker and limb segment. This information can then be used to transform tracker data into limb segment data.

Figure 20 and (eq. 4.1) are reproduced from Chapter IV and provided below as Figure 22 and (eq. 5.1). They describe the relationships between the world, tracker and limb segment frames of reference.

$$ {}^{w}\mathbf{H}_{l} = {}^{w}\mathbf{H}_{t}\,{}^{t}\mathbf{H}_{l} \qquad \text{(eq. 5.1)} $$

The goal of the calibration process is to determine ${}^{t}\mathbf{H}_{l}$, the homogeneous transformation matrix specifying the limb segment frame relative to the tracker's frame.

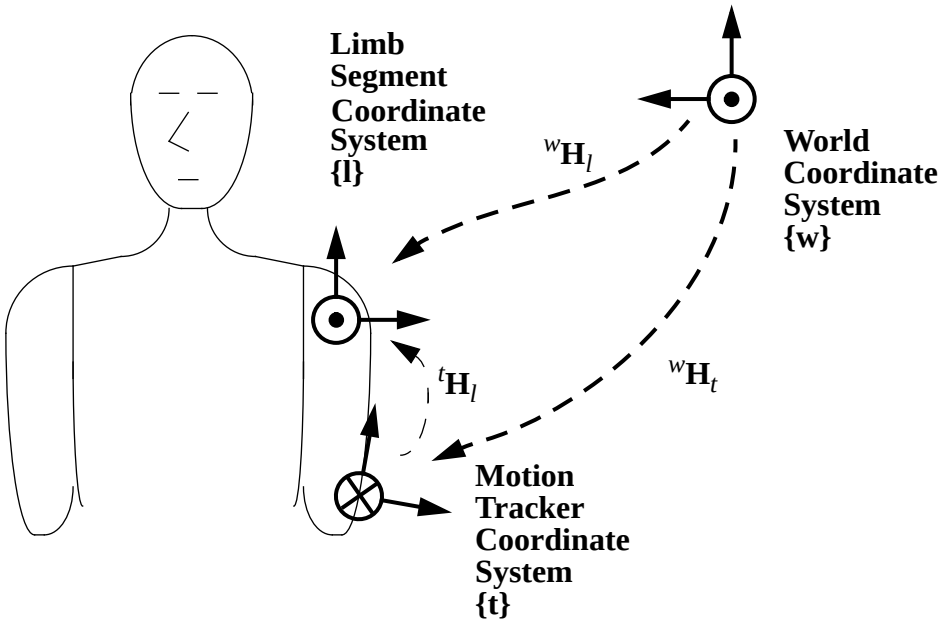

Figure 22: Relationships Between Frames of Reference

This can be found by placing the limb segment in a known reference position (ie. $^{w}\mathbf{H}_{l}$ is known) and measuring tracker position and orientation, $^{w}\mathbf{H}_{t}$, at that time. Then, (eq. 5.2) can be used to compute $^{t}\mathbf{H}_{l}$, the desired constant transformation matrix.

$$(^{w}\mathbf{H}_{t})^{-1}\,{}^{w}\mathbf{H}_{l} = {}^{t}\mathbf{H}_{w}\,{}^{w}\mathbf{H}_{l} = {}^{t}\mathbf{H}_{l} \qquad \text{(eq. 5.2)}$$

1. Angles-only Calibration Technique

If one intends to only use orientation data for determining joint angles, then the calibration process is simplified. In this case only reference orientation is used, instead of orientation and position. The limb segment need only be placed in the specified orientation. Additionally, position elements of the homogeneous transformation matrices given in (eq. 5.1) and (eq. 5.2) may be ignored, giving the following:

$$^{w}\mathbf{R}_{l} = {}^{w}\mathbf{R}_{t}\,{}^{t}\mathbf{R}_{l} \qquad \text{(eq. 5.3)}$$

and

$$({}^{w}\mathbf{R}_{t})^{-1}\,{}^{w}\mathbf{R}_{l} = {}^{t}\mathbf{R}_{w}\,{}^{w}\mathbf{R}_{l} = {}^{t}\mathbf{R}_{l} \qquad \text{(eq. 5.4)}.$$

For the angles-only implementation, the reference orientation is that shown in Figure 8. In order to position himself for calibration, the user needs to match his body position as closely as possible to the reference position. This is done by standing straight up and down and placing the right arm straight down with the thumb forward and elbow locked. The user must stand facing in the direction of the world coordinate system x-axis with his right shoulder aligned with the world coordinate system positive y-axis as shown in Figure 23. Calibration occurs after sensors are initialized on program start-up. The user is prompted from the screen to position himself and is given three seconds to do so after the "enter" button is pushed. At that time, sensor data is taken and the calibration transformation matrices are computed in the **calibrate( )** function of the Body class. These matrices are then stored for use in their corresponding Body class data members.

2. Position Calibration Technique

The position calibration technique is similar to the angles-only technique with the exception that initial tracker and limb segment positions relative to each other and the world coordinate system becomes important. The calibration position used, however, is identical to that used in the orientation calibration technique. Figures 23 and 24 show the required positioning for calibration of the torso and upper arm sensors used to investigate position tracking techniques. As with the orientation calibration technique, the user must stand facing in the direction of the world coordinate system x-axis with his right shoulder aligned with the world coordinate system positive y-axis as shown in Figure 23. Additionally, the torso sensor is mounted such that it is at the same world coordinate z-coordinate as the shoulder and clavicle joints. The upper arm sensor is mounted on the arm so that it is at the same world coordinate system x-coordinate as the shoulder and clavicle joints. Positioning the body and sensors in this manner allows reported sensor positions to

be used in a simple and direct manner for determining joint positions. A more detailed explanation follows.

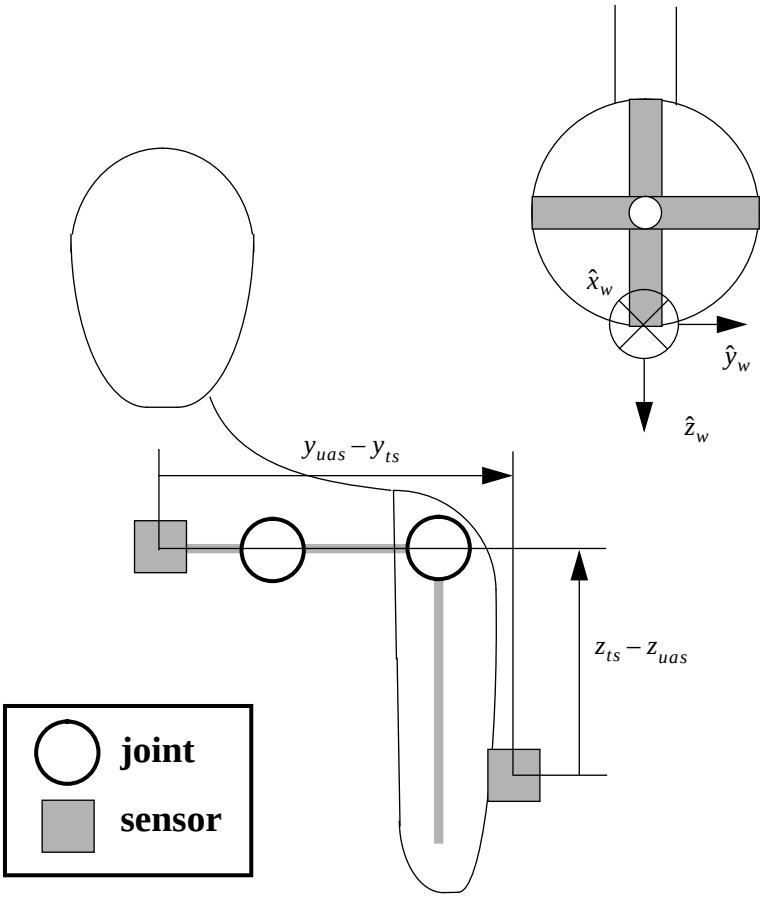

Figure 23: Tracker Position Calibration - Front View

The purpose of position calibration is to determine the position of a joint relative to the sensor which will be tracking it. In the position tracking application investigated here, this means determining the constant vectors ${}^{ts}p_{posit16}$ and ${}^{uas}p_{posit18}$ (see Figure 21). These vectors are then inserted into 4x4 unit matrices to create ${}^{ts}\mathbf{H}_{posit16}$ and ${}^{uas}\mathbf{H}_{posit18}$ which are used in (eq. 4-18) and (eq. 4-19) to convert reported sensor location to joint location. Once

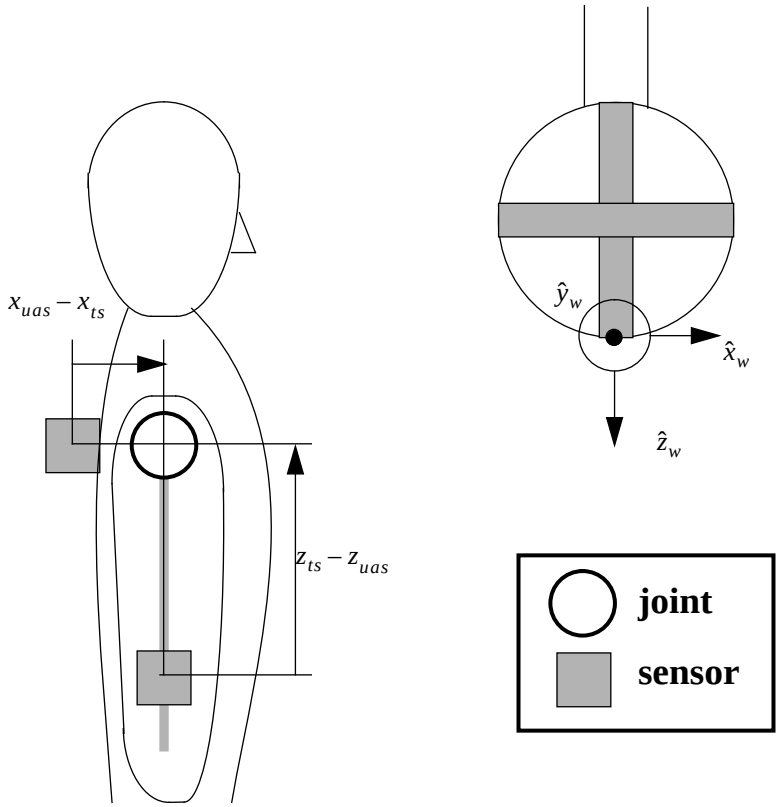

Figure 24: Tracker Position Calibration - Side View

joint locations are known, the inverse kinematic computations described in Chapter IV can be accomplished.

Measurements to determine ${}^{ts}p_{posit16}$ and ${}^{uas}p_{posit18}$ are first taken with reference to the world coordinate system. The vector from the torso sensor to the clavicle and from the upper arm sensor to the shoulder joint are measured with reference to the world coordinate system. This is done by using the sensors themselves and by manual means. These vectors can be called ${}^{tx}p_{ts\_to\_posit16}$ and ${}^{tx}p_{uas\_to\_posit18}$ since they are measured relative to the world coordinate system or the transmitter frame of reference. Figures 23 and 24 show how the various components of these vectors are determined. This is done as follows:

- ${}^{tx}p_{ts\_to\_posit16}$ x-coordinate - determined by taking the difference between the reported x-coordinate of the upper arm sensor and the reported x-coordinate of the torso sensor.

- ${}^{tx}p_{ts\_to\_posit16}$ y-coordinate - this measurement is taken manually by estimating the center of rotation of the clavicle joint and measuring the distance to it in the y-direction.
- ${}^{tx}p_{ts\_to\_posit16}$ z-coordinate - since the torso sensor is at the same z-coordinate as the clavicle joint, this coordinate is set to zero.
- ${}^{tx}p_{uas\_to\_posit18}$ x-coordinate - since the upper arm sensor is at the same x-coordinate as the shoulder joint, this coordinate is set to zero.
- ${}^{tx}p_{uas\_to\_posit18}$ y-coordinate - taken manually by estimating the center of rotation of the shoulder joint and measuring the distance from the spine to it in the y-direction. The difference between the reported y-coordinate of the torso sensor and the reported y-coordinate of the upper arm sensor is then subtracted from this distance and assigned a negative sign.
- ${}^{tx}p_{uas\_to\_posit18}$ z-coordinate - determined by taking the difference between the reported z-coordinate of the torso sensor and the reported z-coordinate of the upper arm sensor.

The manual measurements associated with the y-coordinates of each of these vectors were taken and hard-coded into the calibration routine for a single user. This can be seen in the code of Appendix D and was done for testing only. The manual measurements taken here must correspond with those taken for sizing the model to the user. In particular, the measurements and computations which result in the spine-to-shoulder-joint length and the spine-to-clavicle-joint length (the link_6 length) must correspond to those input or computed when the kinematic model is sized. Model sizing is discussed in the next section of this chapter.

Once ${}^{tx}p_{ts\_to\_posit16}$ and ${}^{tx}p_{uas\_to\_posit18}$ have been determined, it is necessary to convert these vectors into ${}^{ts}p_{posit16}$ and ${}^{uas}p_{posit18}$ respectively. This is done using the following equations:

$${}^{ts}p_{posit16} = ({}^{tx}\mathbf{R}_{ts})^{-1}\,{}^{tx}p_{ts\_to\_posit16} \qquad \text{(eq. 5.5)}$$

$${}^{uas}p_{posit18} = ({}^{tx}\mathbf{R}_{uas})^{-1}\,{}^{tx}p_{uas\_to\_posit18} \qquad \text{(eq. 5.6)}$$

where ${}^{tx}\mathbf{R}_{ts}$ and ${}^{tx}\mathbf{R}_{uas}$ are taken from ${}^{tx}\mathbf{H}_{ts}$ and ${}^{tx}\mathbf{H}_{uas}$ as reported by the torso and upper arm sensors at the time of calibration. As previously stated, the constant vectors ${}^{ts}p_{posit16}$ and

${}^{uas}p_{posit18}$ are then inserted into 4x4 unit matrices to create ${}^{ts}\mathbf{H}_{posit16}$ and ${}^{uas}\mathbf{H}_{posit18}$ and the calibration process is completed.

B. SIZING THE MODEL

In order to ensure that the user can effectively interact with his virtual environment, the kinematic model is scaled to the user's dimensions. This results in a virtual human who is proportional in dimensions to its user. That way, when the user touches his right shoulder with his left finger tips, the virtual human does also. Thus, scaling the user's dimensions into the virtual environment is desired to enhance the perception that the user himself is immersed in the virtual environment. It also means that each virtual human can be scaled appropriately to objects found in the virtual environment, objects with which the user may want to interact.

The method for sizing the model to the user is straight-forward. Prior to system initialization, the user's measurements in centimeters are taken. These measurements include: 1) the spine to shoulder joint length, 2) the upper arm length, 3) the lower arm length, and 4) the hand length. Measurements are taken by estimating the center of the joints in question. During calibration, the user is prompted for these measurements in the **calibrate( )** function of the Body class.

Following input, the code shown in Figure 25 is executed to set the appropriate link lengths and joint offsets to size the model. Notice that the input spine to shoulder length is used to set the link lengths for links 6, 8, and 17. These correspond to the distances between the spine and clavicle joint on each side, and the left and right clavicle link lengths respectively. To do this, the clavicle position is arbitrarily assumed to be approximately 36% of the distance from the spine to shoulder away from the spine. The clavicle lengths then become 64% of the manual measurement. The code is written so that setting the link length for link_6 automatically causes the inboard link lengths for link_7 and link_{16} to be set

accordingly. This occurs in the **set_link_length( )** member function of the Upperbody class which is called by the Body class member function of the same name. The link length of $link_3$ and joint displacement of $joint_{16}$ are hard-coded to arbitrarily set the size of the back, since tracking actually occurs at the torso sensor located between the shoulder blades. Setting the $joint_{16}$ displacement in the Upperbody class member function **set_joint_displacement( )** automatically sets the $joint_6$ displacement and vice-versa, since the two are identical.

```
// set upperbody dimensions to that of the user
set_link_length(3, 21.0);
set_link_length(6, 0.36 * spine_shoulder_length);
set_joint_displacement(16, 26.0);
set_link_length(17, 0.64 * spine_shoulder_length);
set_link_length(8, 0.64 * spine_shoulder_length);
set_link_length(20, uarm_length);
set_link_length(11, uarm_length);
set_link_length(21, larm_length);
set_link_length(12, larm_length);
set_link_length(24, hand_length);
set_link_length(15, hand_length);
```

Figure 25: Excerpt from Body Class Calibrate Member Function

Admittedly, the method chosen here for reducing the number of measurements will in some cases result in an unacceptable match between the model and user with respect to clavicle joint location. A better method would be to submit individual measurements for the joint displacement and clavicle length, rather than linking the two to a single input. Since the clavicle joint is not a rotary joint which can be specifically located, approximating its location is necessary in any event. The method chosen here was found to be acceptable for matching the model to the user in a manner that permitted gross motor control of the clavicle for testing purposes.

C. SUMMARY

Calibration of the tracking hardware and software model were discussed in this chapter. The relationships between the world coordinate system, tracker coordinate systems and coordinate systems attached to the skeletal structure were presented. Tracker positioning for calibration of angle and position tracking methods was also discussed. The chapter concluded with an explanation of how the software model can be scaled to the user. The next chapter presents observations concerning the effectiveness of the interface as a whole.

VI. RESULTS

The interface described in this thesis was investigated with regards to the three major goals for this project discussed in Chapter I. First, the interface needed to be effective in driving realistic and reasonably accurate movement of the virtual human. Second, the interface needed to be efficient enough to ensure that all actions commanded occur in real-time. Third, the interface was to be intuitive and easy to use. The investigation was more qualitative than quantitative due to a lack of hardware for accurately measuring the user's movements (determining truth values) for comparison with movements the system actually produced. Despite this, there are some observations and conclusions worth noting. This chapter provides observations concerning each of the thesis goals. The next chapter presents conclusions drawn from these observations and discusses future work which can be done.

A. REPRESENTATION OF HUMAN MOTION

The system is required to replicate the user's movements in such a way that the motion is realistic and accurate. In this section, realism and accuracy are discussed concerning the angles-only implementation and the position tracking implementation. Results described here can be found in Appendix E, a demonstration video of the angles-only and position implementations of this interface.

1. Angles-only Implementation

The realism required should be sufficient to make the user and any other observers believe that the virtual human represents an actual human in motion. If measured by this standard, then the angles-only implementation can be termed successful. Figure 26 shows operation of the angles-only tracking implementation. The system tracks thirteen degrees of freedom, including a six degree of freedom torso (position and orientation) and one

Figure 26: Angles-only Tracking Implementation in Operation

seven degree of freedom arm. The virtual human's movements smoothly track the user's movements. They are easily identifiable as being those of an actual human by virtue of the subtle coordination of movements typically associated with certain human behavior. An example is walking. The lean of the figure's torso, bounce in the figure's walk and corresponding stretch of the arm give a life-like and realistic flavor to the replicated motion.

The accuracy of motion replication is more difficult to judge. In attempting to determine the accuracy of the representation, one must first measure the actual posture (including such metrics as end-effector position and joint positions and angles) and compare this posture against the posture rendered at sampled times throughout the

movement. This is a large and difficult undertaking. It involves double instrumentation and tracking of the user in a manner that insures one tracking method does not interfere with the other. For example, mechanical trackers which are made of metal may interfere with the electromagnetic trackers used in the interface. For this reason, the methods for determining the accuracy of the system were more subjective and narrow in their scope.

There were two types of tests used to determine in a limited fashion whether or not the system was accurately tracking the user's movements. Both involved statically positioning the user. The first involved observing correlation of joint angles when placed in either 90 or 180 degree positions. In general, this test resulted in visually acceptable angles being generated. The second test observed the ability to position one body part on another, such as placing the finger tips on the shoulder or touching the chest with the palm of the hand. The results of these tests, while not perfect, were encouraging. Several corrections to the system were made (described below), resulting in somewhat improved performance. A more detailed discussion follows.

The accuracy of the duplicated motion is dependent on several factors. The first is the ability of the electromagnetic trackers themselves to accurately report their position and orientation throughout the working volume. Polhemus reports that the static accuracy for a receiver positioned within 76 cm (30 in) of the transmitter is 0.15 degrees RMS for receiver orientation [POLH93]. Operation beyond this range or in the presence of ferromagnetic interference will reduce accuracy. Since the transmitter's antennas are mounted to the ceiling and the user moves freely about it, actual operation is often beyond this range. Also, the environment in which the device was tested had numerous objects which could cause ferromagnetic interference, including metal light fixtures, a metallic drop floor and the computer and monitor used to run the interface. If a hypothetical case is considered where all four sensors are within 76 cm of the transmitter's antennas and the user's spine to finger tip length is 90 cm, and if the arm and hand are extended straight out

to the side and all four sensors are off by 0.15 degrees in the same direction, then the hand's orientation would still only be off by 0.15 degrees with an accumulated position error of approximately 0.24 cm at the finger tips. However, given the working environment and test results, it is likely that the errors are greater.

Livingston and State investigated the registration of electromagnetic trackers at the University of North Carolina [LIVI96]. Though they used Ascension Technologies *Flock of Birds* trackers, their work illuminates the inaccuracies inherent in today's electromagnetic trackers and shows the difficulties which must be overcome to improve registration of these trackers. Their orientation error data for a single *Flock of Birds* sensor (less sensitive to ferromagnetic interference than Polhemus) was taken in a spherical working volume of two meter radius in a similar environment. Using 11,419 measurements, the data shows an average error of 3.34 degrees with a standard deviation of 1.31 degrees and recorded minimum and maximum errors of 0.13 and 20.46 degrees respectively. If we apply the average orientation error in their findings to the example above, then the accumulated position error of the finger tips now becomes approximately 5.28 cm. This error is noticeable in some applications. It may explain in part why positioning the hand over or touching other body parts sometimes resulted in a perceptible disconnect between reality and the virtual representation. On the other hand, it can be difficult to observe only three degrees of error at any given joint angle. This may explain why errors were not as noticeable using the 90 degrees joint angle test.

Another cause of inaccuracies is the motion of a tracker relative to the underlying skeletal structure which it is tracking. During testing, it became evident that this was in fact the case for the torso sensor. Motion of either clavicle would cause the harness on which the sensor was mounted to move, moving the sensor relative to the spine. This was due in large part to the motion of the shoulder itself and the shoulder blades around and on which the straps of the harness rest. Unfortunately it is difficult to keep the clavicle immobile

while moving the arm. This resulted in a noticeable tilt in the body position from the vertical when the arm was raised. The rest of the tracked limb segments (arm and hand) retain the proper orientation relative to world coordinates due to the manner in which joint angles are calculated. If the torso sensor is held in position on the spine by another individual, then the represented body orientation and posture is much closer to reality. To improve torso sensor tracking a new harness was designed. The new harness utilized a single strap suspender design with shoulder straps splitting high up the back and running as close as possible to the neck. The straps were then crossed in front and attached to a belt wrapped around the torso just below the waist. This helped avoid much of, but not all, the movement of the sensor when the clavicles were used. Similar problems were not observed with respect to the remaining limb segment trackers.

Infidelity in the model chosen can also cause accuracy problems. Since the clavicle was modeled but not tracked, the effect is as though it was not modeled at all. As previously mentioned, it is difficult to move the arm without moving the clavicle associated with it. For movements such as scratching one's back by reaching up and over the shoulder, large amounts of clavicle angles are induced (depending on the individual user). This results in a loss of correlation between reality and the representation. In effect, the reachable workspace of the user extends beyond what is reachable for the model. For scratching the back, the result is that the icon's hand hangs above where the user's hand is actually positioned. This problem can only be corrected by tracking the clavicle motion and applying it to the model.

A second problem with the model was observed regarding the imposition of joint limits. The joint limits shown in Table 3 were arrived at by observing apparent limitations in movement of the human joints in the direction of the modeled joints. In fact, only a few restricted movements were observed in isolation from other movements. The range of movements available to the user, a result of combining motion around all three axis of

rotation, require these model limits be expanded. This was discovered when the icon's posture snapped from various positions to others during testing. Removing most of the joint limits (as has been done in Appendix A) resulted in elimination of the problem. Since the physical joint limits of the user himself will prevent unrealistic articulation of the figure, it would seem there is no need for joint limits in the model.

The method chosen to handle singularities in the system, avoidance, does not result in noticeable inaccuracies. One might expect that skipping over singularities might result in a discontinuity in motion. Attempts to place the system in a singularity or move the system through a singularity show no difference from other types of motion.

Inaccuracies in the method chosen to calibrate the system can cause problems. During testing of the elbow joint, it was noticed that the elbow lagged the actual elbow joint angle by approximately 10 degrees. It was discovered that this was a result of assuming 0 degrees joint angle at the time of calibration, when in fact this is not possible for most users without hyperextension of the elbow. This resulted in a computed elbow joint angle which was always less than the actual joint angle by an amount equal to the difference between actual and reference positions at the time of calibration. To correct this, the reference position was changed to a position where the elbow is bent 90 degrees with the arm forward and thumb pointed up. This placed the lower arm and hand parallel to the world coordinate system x-axis and required that the two reference calibration matrices associated with the lower arm and hand be modified in the code. The problem was no longer observed. It should be noted, however, that accuracy is still effected by how closely the user can position himself to the reference position.

Accuracy of motion replication involves timing also. It is not enough that static positions be replicated accurately. The goal is for the motion to be replicated with the same rates and accelerations. If the system exhibits minimal lag and can position the virtual human in real-time, then the rates and accelerations displayed will be close to that of the

actual user. An attempt at quantifying system lag was not made for lack of suitable measuring equipment. Subjectively, there is a perceptible lag in the virtual human's movements, but it is not large for even the fastest of arm movements. Polhemus provides the latency of its tracker as four ms from the center of the receiver measurement period to the beginning of transfer from output port [POLH93]. It is likely that most of the perceived lag, however, is a result of overhead due to transfer of data from the output port to the SGI and device driver buffers, and subsequent rendering. When compared with the interface developed by McMillan [MCMI96b], lag is much improved. This is probably due to the fact that McMillan's interface used a more complex model (*Jack*) and worked in a networked virtual environment (NPSNET). Excessive lag in this case, resulted in a loss of fluidity in the rendered motion. This highlights the need for identifying system choke points in order to ensure maximum efficiency when working in the networked environment.

2. Position Implementation

From the outset, it became apparent that it would be difficult to use the Polhemus electromagnetic trackers for position tracking. It was possible to observe control of the virtual human's clavicle in all four directions, but only if the user's position remained very close to the calibration position. The representation exhibits constant jitter due to sensor drift. Movement away from the calibration position results in a lack of controllability due to poor tracking system registration. Polhemus reports a static position accuracy of 0.08 cm RMS for x, y or z receiver position when within 76 cm of the transmitter's antennas [POLH93]. In a simple test, two sensors were attached 30 cm apart to a plastic ruler. The magnitude of the position vector between them was computed and output to the screen. The sensors where moved throughout a one meter cubed workspace corresponding to the area in which the clavicle was tested. Readings changed rapidly and varied between

20 cm and 54 cm. Livingston and State's data for raw position error, taken concurrent with their measurements for orientation error, show an average error of 5.69 cm with a standard deviation of 4.55 cm and minimum and maximum errors of .11 and 32.17 cm respectively [LIVI96]. Two sensor positions are required to track the clavicle. This being the case, it is clear that these errors are too large to allow accurate position tracking of a nominal 14 cm long clavicle.

B. EASE OF USE

The interface is easy to set-up and use. The user must have four measurements taken, including spine to shoulder length, upper and lower arm lengths, and the hand length. After donning the equipment and turning on the *Fastrak* system, the program can be run. The user inputs his measurements and then stands in the calibration position. Pressing the "enter" key causes a three second delay, after which calibration measurements are taken and tracking commences. Following this, it only takes a few seconds to display the replicated image. The entire process takes approximately five minutes.

Control of the figure is as easy as controlling one's own body. The intuitive nature of the interface is born out by the fact that small children can successfully use it when told to position the figure in a variety of postures. Sensor mounting does not restrict movement, if care is taken to make sure the upper arm sensor is not mounted too close to the elbow. Sensor wiring is perhaps the most encumbering aspect of the interface. Routing sensor wiring up the arm, through the arm and torso sensor straps, and down the back helps to avoid entanglement, though the user must be mindful of wiring that trails back to the *Fastrak* control box. Sensor wiring and transmitter range restrict use to approximately 10 -12 feet from the transmitter's antenna. Within this workspace, however, the user is relatively free to move about.

VII. SUMMARY AND CONCLUSIONS

A. SUMMARY

This thesis addressed the problem that virtual environments (VE's) do not possess a practical, intuitive, and comfortable interface that allows a user to control a virtual human's movements in real-time. The approach was to develop an interface for the upper body, since it is through this part of users' anatomy that they interact most with their environment. Implementation included construction of a 24 DOF Danevit-Hartenberg kinematic model of the upper body. The model is manipulated in real-time using orientation data from electromagnetic motion tracking sensors placed on the user.

Electromagnetic trackers were chosen because of their 6 DOF tracking capability, availability, and low cost. Their small size makes them easy to attach to the user. Calibration of these sensors is a straight-forward process. The user simply positions himself in a reference position for a single set of sensor readings. The device takes approximately one sixth the time to don and calibrate as do mechanical interfaces with similar capability.

Research resulted in an interface that is easy to use and allows its user to interact with a VE. The device tracks thirteen degrees of freedom. Upper body position is tracked, allowing the users to move through the VE. When using the device, the user has the feeling of being immersed in the VE. The interface can be used for a variety of applications which do not require higher levels of precision.

B. CONCLUSIONS

The electromagnetic tracking systems available today lack sufficient accuracy and registration to enable their use as true six DOF trackers. It was hoped that by using a traditional kinematic notation together with six degree of freedom sensors, a smaller number of sensors and less encumbering system would result. While this may be true, in many applications it would be desirable to mount a six DOF tracker on each limb, rather

than reduce the number of trackers. This enables one to use the redundant tracking data to improve accuracy for precision applications, such as sighting and firing a rifle in the virtual environment. Accurate and well registered six DOF electromagnetic trackers could be used in many applications in which the current trackers cannot be used. This would be of great benefit to the designers and users of virtual environments.

One method of improving the registration of electromagnetic trackers is to calibrate the device within a specified workspace and provide a look-up table for error correction during operation. This is exactly the work being undertaken by Livingston and State at the University of North Carolina [LIVI96]. Their experimentation with a Faro Metrecom IND-1 mechanical tracker and *Flock of Birds* sensor has resulted in a method which reduced the average position error by 78% and the average orientation error by 40%. They found, however, that orientation errors depend not only on the tracker's position, but also on its orientation. Their lack of success in reducing orientation error is attributed to their original assumption to the contrary. A 6D look-up table is therefore required for orientation. This makes a look-up table calibration method impractical for orientation. However, if the look-up table method is applied to position data, it should reduce errors enough to significantly improve registration. This could make electromagnetic trackers usable as a true six DOF trackers for some applications, including this one.

The usability of electromagnetic trackers for military applications is greatly reduced by several aspects of their design. First, their sourced nature requires the user to remain within range of the transmitter. In some combat training scenarios this would be a severe restriction. Second, their susceptibility to ferromagnetic interference can greatly impede their use in the military environment. Care must be taken to avoid operation near ferromagnetic objects, such as rifles, certain types of combat gear, electronic equipment, and the metal bulkheads of ships, tanks and other vehicles or simulators. Finally, the system used here was tethered. This means that large or cluttered working volumes can be

difficult to work in. Military applications require large working volumes, greater freedom of motion, and freedom from ferromagnetic interference. This makes sourceless and untethered systems that are robust in an environment high in electromagnetic interference very desirable.

The design of this interface, as with any other engineered product, reflects trade-offs between achievable capabilities. For example, by making the system easy to calibrate some precision is sacrificed. Also, by reducing the complexity of the model for increased speed and efficiency, some accuracy in replicating the user's movements or simulating his reaction to the environment is lost. For example, a dynamic model and force feedback could be used to create a more acceptable training environment, but at a cost of much greater complexity and system overhead. The question is whether an acceptable balance can be achieved between trade-offs *when considering the overall purpose of the system*. One must recognize, however, that limitations of available technology may preclude successful achievement of all design goals.

The success of this interface is application dependent. Consider that success is related to the ability of the user to use the interface to accomplish tasks and learn in the virtual environment. The angles-only implementation of the system is usable for tasks which can be accomplished with only gross motor control. These include various forms of locomotion and dance, use of arm signals, communicative interaction with virtual devices, moving virtual objects, and playing games that require lower amounts of coordination, such as virtual handball with a large or slower moving ball. It is unlikely that this interface could be used for applications requiring higher levels of precision, such as aiming and firing a rifle. It remains to be seen whether the interface can be used in the networked virtual environment and for what purposes.

C. FUTURE WORK

The angles-only interface can be improved in several basic ways. Additional sensors could be added for articulation of the left arm, and the system could be modified to allow articulation of the head. The ability to provide the user a view from the eyes of the virtual human is also desirable. Finally, the rendered appearance of the model can be improved by incorporating one of several graphic models currently available in academia and the commercial sector.

Additional testing of the interface is necessary to determine its strengths, weaknesses, and suitability for the networked virtual environment. Though a major undertaking, a quantitative evaluation of the accuracy with which the figure can be positioned is desirable. It is also important to determine the lag in the interface. Results of these tests could be used to identify possible improvements. More subjective testing of the ability of users to accomplish various tasks should also be conducted. Following these quantitative and qualitative tests, the interface could be implemented in a networked virtual environment and the tests repeat.

The issue of using electromagnetic tracker position data should be revisited. A look-up table calibration method such as the one described in [LIVI96] and a suitable filter for reducing the jitter associated with sensor drift could be implemented. The interface could then be modified for increased accuracy or for use with fewer sensors. Such a change could result in significantly improved application of the interface.

Finally, research in other areas of modeling and tracking may result in more effective interfaces. For example, by modeling the body using a quaternion notation one can eliminate singularities [COOK92]. Rate tracking through positions that otherwise would have been singularities can occur. Suitable quaternion filters can be designed to combine tracking inputs and enhance the output of the interface [BACH96b].

Alternate new tracking technologies should also be investigated. Spread spectrum electromagnetic tracking and artificial vestibular (inertial) systems are particularly promising. The spread spectrum system would enhance accuracy, resolution, response, robustness, and sociability over current electromagnetic systems. An artificial vestibular system is the only sourceless system, allowing for excellent robustness and sociability. Perhaps a combination of tracking technologies will result in the best system, with advantages of one technology compensating for disadvantages of another.

This thesis provides a starting point for those interested in developing better interfaces for users of Virtual Environments. It can be found on the Internet at http://www-npsnet.cs.nps.navy.mil/npsnet/publications.html. Code used in this thesis can be accessed via anonymous file transfer protocol (FTP) at ftp://ftp-npsnet.cs.nps.navy.mil/pub/skopowski/ArticulatedHuman.tar.Z.

APPENDIX A: ANGLE TRACKING SOFTWARE

link.h **1**

```
// *******************************************************
// FILENAME:  link.h
// PURPOSE:  declarations for the link class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition and some functions of the link class
// *******************************************************

#ifndef LINK_H
#define LINK_H

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>



class Link
{
   public:
      Link(double, double, double, double, double, double,
           float = 0.0, float = 0.0, float = 0.0, float = 0.0);

//-----------------------------------------------------------
// Function: rotate (double angle)
// Purpose: set the link's joint angle
//-----------------------------------------------------------
      void rotate (double angle)
      {
          if (angle < min_joint_angle){
             angle = min_joint_angle;
          }
          if (angle > max_joint_angle){
             angle = max_joint_angle;
          }
          joint_angle = angle;
      }


//-----------------------------------------------------------
// Function: rotate_increment (double increment_angle)
// Purpose: increment the link's joint angle
//-----------------------------------------------------------
      void rotate_increment (double increment_angle)
      {
          double angle = increment_angle + joint_angle;
          rotate (angle);
      }
```

```
//---------------------------------------------------------
// Function: draw()
// Purpose: draw the link in the proper position/orientation
//---------------------------------------------------------
      void draw()
      {
         glTranslated ((GLdouble) inboard_link_length, 0.0, 0.0);
         glRotated ((GLdouble) inboard_twist_angle, 1.0, 0.0, 0.0);
         glTranslated (0.0, 0.0, (GLdouble) joint_displacement);
         glRotated ((GLdouble) joint_angle, 0.0, 0.0, 1.0);
         if (draw_length > 0.0){
            drawDiamond(0.0, 0.0, 0.0, draw_length, draw_width,
                                       draw_depth, draw_offset);
         }
      }


//---------------------------------------------------------
// Function: reset()
// Purpose: reset the link's joint angle to its zero posit
//---------------------------------------------------------
      void reset()
      {
         joint_angle = initial_joint_angle;
      }


//---------------------------------------------------------
// Function: set_joint_displacement(float displacement)
// Purpose: set the link's joint displacement
// Returns: TRUE
//---------------------------------------------------------
      int set_joint_displacement(float displacement)
      {
         joint_displacement = displacement;
         return 1;
      }


//---------------------------------------------------------
// Function: set_inboard_link_length(float length)
// Purpose: set the link's inboard link length
// Returns: TRUE
//---------------------------------------------------------
      int set_inboard_link_length(float length)
      {
         inboard_link_length = length;
         return 1;
      }
```

```
//-----------------------------------------------------------
// Function: set_draw_length(float length)
// Purpose: set the draw_length of the link
// Returns: TRUE
//-----------------------------------------------------------
      int set_draw_length(float length)
      {
         draw_length = length;
         return 1;
      }

//-----------------------------------------------------------
// Function: set_draw_width(float width)
// Purpose: set the draw_width of the link
// Returns: TRUE
//-----------------------------------------------------------
      int set_draw_width(float width)
      {
         draw_width = width;
         return 1;
      }

//-----------------------------------------------------------
// Function: get_draw_offset()
// Purpose: get the draw_offset of the link
// Returns: draw_offset
//-----------------------------------------------------------
      float get_draw_offset()
      {
         return draw_offset;
      }
```

```
   protected:
      double inboard_link_length;
      double inboard_twist_angle;
      double joint_displacement;
      double initial_joint_angle;
      double joint_angle;
      double min_joint_angle;
      double max_joint_angle;
      int number_outboard_links;
      float draw_length;
      float draw_width;
      float draw_depth;
      float draw_offset;

      // utility functions
      void drawDiamond(float x, float y, float z,
              float length, float width, float depth, float offset);

      void computeNormal(const float* a, const float* b, const float* c,
                         float *result);

};
#endif
```

```
// ******************************************************
// FILENAME:  link.cc
// PURPOSE:  function definitions for the link class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: Link constructor and drawDiamond function
// MODIFIED: 9 Mar 96
// ******************************************************

#include "link.h"
#include <math.h>


//-------------------------------------------------------
// Function: Link(double ill, double ita, double jd,
//         : double ija, double min_ja, double max_ja,
//         : float dl, float dw, float dd, float doff)
// Purpose: constructor of the link type
// Returns: link class object
//-------------------------------------------------------
Link::Link(double ill, double ita, double jd, double ija,
           double min_ja, double max_ja,
           float dl, float dw, float dd, float doff)
{
   inboard_link_length = ill;
   inboard_twist_angle = ita;
   joint_displacement = jd;
   initial_joint_angle = ija;
   joint_angle = ija;
   min_joint_angle = min_ja;
   max_joint_angle = max_ja;
   draw_length = dl;
   draw_width = dw;
   draw_depth = dd;
   draw_offset = doff;
}
```

```
//---------------------------------------------------------
// Function: drawDiamond(float x, float y, float z,
//         : float length,float width, float depth,
//         : float offset)
// Purpose: draw a diamond with end at (x,y,z)
//        : use specified parameters
//        : draws diamond along the x-axis
//        : center of diamond can be offset
//---------------------------------------------------------
void Link::drawDiamond(float x, float y, float z,
              float length,
              float width,
              float depth,
              float offset)

// float x,y,z;        end of the diamond in 3-space.


{
   float midpoint;  // x coordinate for waist vertices of the diamond
   float halfwidth; // half the width
   float halfdepth; // half the depth

   float p[6][3];   // array to hold coords for the diamond vertices.
   float n[3];      // array to hold the normal vector


   // Compute the x-axis midpoint.
   midpoint=length*offset;

   // Compute half the dimensions.
   halfwidth = width/2.0;
   halfdepth = depth/2.0;


   // vertices.
   p[0][0]=x;
   p[0][1]=y;
   p[0][2]=z;

   p[1][0]=x+midpoint;
   p[1][1]=y;
   p[1][2]=z+halfdepth;

   p[2][0]=x+midpoint;
   p[2][1]=y+halfwidth;
   p[2][2]=z;

   p[3][0]=x+midpoint;
   p[3][1]=y;
   p[3][2]=z-halfdepth;
```

```
   p[4][0]=x+midpoint;
   p[4][1]=y-halfwidth;
   p[4][2]=z;

   p[5][0]=x+length;
   p[5][1]=y;
   p[5][2]=z;

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.


   // Compute and set normal for the first side
   computeNormal(p[0], p[1], p[2], n);
   glNormal3fv(n);

   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[1]);
      glVertex3fv(p[2]);
   glEnd();

   //glColor3f(0.0, 0.0, 0.0);   // Set the color to black.

   // Compute and set normal for the first side
   computeNormal(p[0], p[2], p[3], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[2]);
      glVertex3fv(p[3]);
   glEnd();

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.

   // Compute and set normal for the first side
   computeNormal(p[0], p[3], p[4], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[3]);
      glVertex3fv(p[4]);
   glEnd();

   //glColor3f(0.5, 0.5, 0.5);   // Set the color to gray.

   // Compute and set normal for the first side
   computeNormal(p[0], p[4], p[1], n);
   glNormal3fv(n);
```

```
   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[4]);
      glVertex3fv(p[1]);
   glEnd();

   //glColor3f(0.5, 0.5, 0.5);   // Set the color to gray.

   // Compute and set normal for the first side
   computeNormal(p[5], p[2], p[1], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[2]);
      glVertex3fv(p[1]);
   glEnd();

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.

   // Compute and set normal for the first side
   computeNormal(p[5], p[3], p[2], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[3]);
      glVertex3fv(p[2]);
   glEnd();

   //glColor3f(0.0, 0.0, 0.0);   // Set the color to black.

   // Compute and set normal for the first side
   computeNormal(p[5], p[4], p[3], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[4]);
      glVertex3fv(p[3]);
   glEnd();

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.

   // Compute and set normal for the first side
   computeNormal(p[5], p[1], p[4], n);
   glNormal3fv(n);
```

```
   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[1]);
      glVertex3fv(p[4]);
   glEnd();

   // the diamond is drawn.

}

//-------------------------------------------------------
// Function: computeNormal(const float* a, const float* b,
//         : const float* c, float *result)
// Purpose: compute normal vector to a triangular polygon
//-------------------------------------------------------
void Link::computeNormal(const float* a, const float* b, const float* c,
                         float *result)

{
   float x[3];
   float y[3];
   float magnitude;

   // compute the first vector

   x[0] = b[0] - a[0];
   x[1] = b[1] - a[1];
   x[2] = b[2] - a[2];

   // compute the second vector

   y[0] = c[0] - a[0];
   y[1] = c[1] - a[1];
   y[2] = c[2] - a[2];

   // compute the cross product vector

   result[0] = x[1] * y[2] - x[2] * y[1];
   result[1] = x[2] * y[0] - x[0] * y[2];
   result[2] = x[0] * y[1] - x[1] * y[0];

   //normalize the result

   magnitude = sqrt(result[0] * result[0] + result[1] * result[1] +
                    result[2] * result[2]);

   result[0] = result[0]/magnitude;
   result[1] = result[1]/magnitude;
   result[2] = result[2]/magnitude;


}
```

```
// ******************************************************
// FILENAME:  link1.h
// PURPOSE:  declarations for the link1 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link1 class
// ******************************************************

#ifndef LINK1_H
#define LINK1_H

#include "link.h"



class Link1 : public Link
{
   public:
      Link1() : Link ( 0.0, 0.0,  0.0, 0.0,
                       -90.0,  90.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link2.h
// PURPOSE:  declarations for the link2 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link2 class
// ******************************************************

#ifndef LINK2_H
#define LINK2_H

#include "link.h"



class Link2 : public Link
{
   public:
      Link2() : Link ( 0.0, 90.0,  0.0, -90.0, -270.0,
                       30.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link3.h
// PURPOSE:  declarations for the link3 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link3 class
// ******************************************************

#ifndef LINK3_H
#define LINK3_H

#include “link.h”



class Link3 : public Link
{
   public:
      Link3() : Link ( 0.0, 90.0,  0.0,   0.0,
                      -75.0,  75.0, 17.5, 10.5, 6.5, 0.5)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link4.h
// PURPOSE:  declarations for the link4 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link4 class
// ******************************************************

#ifndef LINK4_H
#define LINK4_H

#include “link.h”



class Link4 : public Link
{
   public:
      Link4() : Link ( 17.5, 90.0,  0.0,  90.0,
                       0.0, 180.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link5.h
// PURPOSE:  declarations for the link5 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link5 class
// ******************************************************

#ifndef LINK5_H
#define LINK5_H

#include “link.h”



class Link5 : public Link
{
   public:
      Link5() : Link (  0.0, 90.0,  0.0,  90.0,
                        0.0, 180.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link6.h
// PURPOSE:  declarations for the link6 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link6 class
// ******************************************************

#ifndef LINK6_H
#define LINK6_H

#include "link.h"

class Link6 : public Link
{
   public:
      Link6() : Link ( 0.0, 90.0,  0.0, 180.0,
                       135.0, 225.0, 28.0, 13.0, 6.5, 0.8)
      { }

//-------------------------------------------------------
// Function: draw()
// Purpose: draw the link in the proper position/orientation
//-------------------------------------------------------
      void draw()
      {
         // draw the torso
         glTranslated ((GLdouble) inboard_link_length, 0.0, 0.0);
         glRotated ((GLdouble) inboard_twist_angle, 1.0, 0.0, 0.0);
         glTranslated (0.0, 0.0, (GLdouble) joint_displacement);
         glRotated ((GLdouble) (joint_angle - 90.0), 0.0, 0.0, 1.0);
         drawDiamond(0.0, 0.0, 0.0, draw_length, draw_width,
                                    draw_depth, draw_offset);

         // draw the head
         glRotated ((GLdouble) 45.0, 1.0, 0.0, 0.0);
         drawDiamond(draw_length, 0.0, 0.0, 10.0, 10.0, 10.0, .8);
         glRotated ((GLdouble) -45.0, 1.0, 0.0, 0.0);

         // draw the nose
         glPushMatrix();
         glTranslated((draw_length + 5.0), 0.0, 0.0);
         glRotated (90.0, 0.0, 1.0, 0.0);
         drawDiamond(0.0, 0.0, 0.0, 6.5, 3.0, 3.0, .2);
         glPopMatrix();
         glRotated ((GLdouble) 90.0, 0.0, 0.0, 1.0);
      }

   private:
};

#endif
```

```
// ******************************************************
// FILENAME:  link7.h
// PURPOSE:  declarations for the link7 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link7 class
// ******************************************************

#ifndef LINK7_H
#define LINK7_H

#include “link.h”



class Link7 : public Link
{
   public:
      Link7() : Link ( 7.5, 90.0, 22.5,   0.0,
                      -30.0,  40.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link8.h
// PURPOSE:  declarations for the link8 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link8 class
// ******************************************************

#ifndef LINK8_H
#define LINK8_H

#include “link.h”



class Link8 : public Link
{
   public:
      Link8() : Link ( 0.0, 90.0,  0.0,   0.0,
                      -20.0,  50.0, 12.5, 6.5, 6.5, 0.5)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link9.h
// PURPOSE:  declarations for the link9 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link9 class
// ******************************************************

#ifndef LINK9_H
#define LINK9_H

#include “link.h”



class Link9 : public Link
{
   public:
      Link9() : Link ( 12.5, 90.0,  0.0,  90.0,
                       -360.0, 360.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link10.h
// PURPOSE:  declarations for the link10 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link10 class
// ******************************************************

#ifndef LINK10_H
#define LINK10_H

#include “link.h”



class Link10 : public Link
{
   public:
      Link10() : Link (0.0, 90.0,  0.0,  90.0,
                       -360.0, 360.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link11.h
// PURPOSE:  declarations for the link11 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link11 class
// ******************************************************

#ifndef LINK11_H
#define LINK11_H

#include “link.h”



class Link11 : public Link
{
   public:
      Link11() : Link ( 0.0, 90.0,  0.0,   0.0,
                       -360.0, 360.0, 22.5, 10.0, 10.0, 0.2)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link12.h
// PURPOSE:  declarations for the link12 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link12 class
// ******************************************************

#ifndef LINK12_H
#define LINK12_H

#include “link.h”



class Link12 : public Link
{
   public:
      Link12() : Link ( 22.5, 90.0,  0.0,   15.0,
                        0.0, 170.0, 25.5, 9.0, 9.0, 0.2)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link13.h
// PURPOSE:  declarations for the link13 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link13 class
// ******************************************************

#ifndef LINK13_H
#define LINK13_H

#include “link.h”



class Link13 : public Link
{
   public:
      Link13() : Link ( 25.0, 0.0,  0.0, 0.0,
                        -90.0, 90.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link14.h
// PURPOSE:  declarations for the link14 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link14 class
// ******************************************************

#ifndef LINK14_H
#define LINK14_H

#include “link.h”



class Link14 : public Link
{
   public:
      Link14() : Link ( 0.0, -90.0,  0.0,  -90.0,
                        -180.0,  0.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link15.h
// PURPOSE:  declarations for the link6 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link15 class
// ******************************************************

#ifndef LINK15_H
#define LINK15_H

#include “link.h”



class Link15 : public Link
{
   public:
      Link15() : Link ( 0.0, -90.0,  0.0,  0.0,
                        -90.0, 90.0,  17.0,  3.0, 8.0, 0.2)
   { }

      void draw()
      {
         // draw the hand
         glTranslated ((GLdouble) inboard_link_length, 0.0, 0.0);
         glRotated ((GLdouble) inboard_twist_angle, 1.0, 0.0, 0.0);
         glTranslated (0.0, 0.0, (GLdouble) joint_displacement);
         glRotated ((GLdouble) joint_angle, 0.0, 0.0, 1.0);
         glRotated (-90.0, 0.0, 1.0, 0.0);
         glRotated ( 90.0, 1.0, 0.0, 0.0);
         drawDiamond(0.0, 0.0, 0.0, draw_length, draw_width,
                                       draw_depth, draw_offset);

         // draw the thumb
         glRotated ((GLdouble) -30.0, 0.0, 1.0, 0.0);
         drawDiamond(0.0, 0.0, 0.0, (.75 * draw_length), 3.0, 3.0, .5);
         glRotated ((GLdouble) 30.0, 0.0, 1.0, 0.0);
         glRotated ((GLdouble) 90.0, 0.0, 0.0, 1);
      }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link16.h
// PURPOSE:  declarations for the link16 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link16 class
// ******************************************************

#ifndef LINK16_H
#define LINK16_H

#include “link.h”



class Link16 : public Link
{
   public:
      Link16() : Link (-7.5, 90.0, 22.5, 180.0,
                       140.0, 210.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link21.h
// PURPOSE:  declarations for the link21 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link21 class
// ******************************************************

#ifndef LINK21_H
#define LINK21_H

#include “link.h”



class Link21 : public Link
{
   public:
      Link21() : Link ( 22.5, 90.0,  0.0,   0.0,
                       -360.0,  360.0, 25.5, 9.0, 9.0, 0.2)

      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link23.h
// PURPOSE:  declarations for the link23 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link23 class
// ******************************************************

#ifndef LINK23_H
#define LINK23_H

#include “link.h”



class Link23 : public Link
{
   public:
      Link23() : Link ( 0.0, 90.0,  0.0,  90.0,
                        -180.0, 0.0,  0.0, 0.0, 0.0, 0.0)
      { }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  link24.h
// PURPOSE:  declarations for the link24 class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the link24 class
// ******************************************************

#ifndef LINK24_H
#define LINK24_H

#include “link.h”



class Link24 : public Link
{
   public:
      Link24() : Link ( 0.0, -90.0,  0.0, 0.0,
                        -90.0, 90.0,  17.0, 3.0, 8.0, 0.2)
      { }

      void draw()
      {
         // draw the hand
         glTranslated ((GLdouble) inboard_link_length, 0.0, 0.0);
         glRotated ((GLdouble) inboard_twist_angle, 1.0, 0.0, 0.0);
         glTranslated (0.0, 0.0, (GLdouble) joint_displacement);
         glRotated ((GLdouble) joint_angle, 0.0, 0.0, 1.0);
         glRotated (-90.0, 0.0, 1.0, 0.0);
         glRotated (-90.0, 1.0, 0.0, 0.0);
         drawDiamond(0.0, 0.0, 0.0, draw_length, draw_width,
                                      draw_depth, draw_offset);

         // draw the thumb
         glRotated ((GLdouble) -30.0, 0.0, 1.0, 0.0);
         drawDiamond(0.0, 0.0, 0.0, (.75 * draw_length), 3.0, 3.0, .5);
         glRotated ((GLdouble) 30.0, 0.0, 1.0, 0.0);
         glRotated ((GLdouble) 90.0, 0.0, 0.0, 1);
      }

   private:

};


#endif
```

```
// ******************************************************
// FILENAME:  upperbody.h
// PURPOSE:  declarations for the upper_body class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the upperbody class
// ******************************************************

#ifndef UPPERBODY_H
#define UPPERBODY_H



#include “link1.h”
#include “link2.h”
#include “link3.h”
#include “link4.h”
#include “link5.h”
#include “link6.h”
#include “link7.h”
#include “link8.h”
#include “link9.h”
#include “link10.h”
#include “link11.h”
#include “link12.h”
#include “link13.h”
#include “link14.h”
#include “link15.h”
#include “link16.h”
#include “link21.h”
#include “link24.h”



class Upperbody
{
   private:
      Link1 link1;
      Link2 link2;
      Link3 link3;
      Link4 link4;
      Link5 link5;
      Link6 link6;
      Link7 link7;
      Link8 link8;
      Link9 link9;
      Link10 link10;
      Link11 link11;
      Link12 link12;
      Link13 link13;
      Link14 link14;
```

```
      Link15 link15;
      Link16 link16;
      Link8 link17;
      Link9 link18;
      Link10 link19;
      Link11 link20;
      Link21 link21;
      Link13 link22;
      Link14 link23;
      Link24 link24;

   public:
      Upperbody();

      void rotate (double *);

      void rotate_increment (double *);

      void draw();

      void reset();

      int set_link_length(int, float);

      int set_joint_displacement(int, float);


};


#endif
```

```
// *******************************************************
// FILENAME:  upperbody.cc
// PURPOSE:  functions for the upperbody class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: functions for the upperbody class
// *******************************************************


#include “upperbody.h”    // include the header file for the class

//-------------------------------------------------------
// Function: Upperbody()
// Purpose: constructor of the Upperbody type
// Returns: Upperbody class object
//-------------------------------------------------------
Upperbody::Upperbody()    // constuctor
{ }


//-------------------------------------------------------
// Function: rotate (double angle[25])
// Purpose: set joint angles to the angles specified
//-------------------------------------------------------
void Upperbody::rotate (double angle[25])
{
    link1.rotate(angle[1]);    // set each link’s joint to the new angle
    link2.rotate(angle[2]);
    link3.rotate(angle[3]);
    link4.rotate(angle[4]);
    link5.rotate(angle[5]);
    link6.rotate(angle[6]);
    link7.rotate(angle[7]);
    link8.rotate(angle[8]);
    link9.rotate(angle[9]);
    link10.rotate(angle[10]);
    link11.rotate(angle[11]);
    link12.rotate(angle[12]);
    link13.rotate(angle[13]);
    link14.rotate(angle[14]);
    link15.rotate(angle[15]);
    link16.rotate(angle[16]);
    link17.rotate(angle[17]);
    link18.rotate(angle[18]);
    link19.rotate(angle[19]);
    link20.rotate(angle[20]);
    link21.rotate(angle[21]);
    link22.rotate(angle[22]);
    link23.rotate(angle[23]);
    link24.rotate(angle[24]);
}
```

```
//-------------------------------------------------------
// Function: rotate_increment (double increment_angle[25])
// Purpose: increment joint angles by the angles specified
//-------------------------------------------------------
void Upperbody::rotate_increment (double increment_angle[25])
{

    link1.rotate_increment(increment_angle[1]);  // increment each joint angle
    link2.rotate_increment(increment_angle[2]);
    link3.rotate_increment(increment_angle[3]);
    link4.rotate_increment(increment_angle[4]);
    link5.rotate_increment(increment_angle[5]);
    link6.rotate_increment(increment_angle[6]);
    link7.rotate_increment(increment_angle[7]);
    link8.rotate_increment(increment_angle[8]);
    link9.rotate_increment(increment_angle[9]);
    link10.rotate_increment(increment_angle[10]);
    link11.rotate_increment(increment_angle[11]);
    link12.rotate_increment(increment_angle[12]);
    link13.rotate_increment(increment_angle[13]);
    link14.rotate_increment(increment_angle[14]);
    link15.rotate_increment(increment_angle[15]);
    link16.rotate_increment(increment_angle[16]);
    link17.rotate_increment(increment_angle[17]);
    link18.rotate_increment(increment_angle[18]);
    link19.rotate_increment(increment_angle[19]);
    link20.rotate_increment(increment_angle[20]);
    link21.rotate_increment(increment_angle[21]);
    link22.rotate_increment(increment_angle[22]);
    link23.rotate_increment(increment_angle[23]);
    link24.rotate_increment(increment_angle[24]);
}


//-------------------------------------------------------
// Function: draw()
// Purpose: draw the upperbody
//-------------------------------------------------------
void Upperbody::draw()
{

    link1.draw();    // draw each link, starting at the waist
    link2.draw();
    link3.draw();
    link4.draw();
    link5.draw();
    link6.draw();
    glPushMatrix();  // after drawing upper torso, remember where it was drawn
    link7.draw();    // start drawing left side from the shoulder
    link8.draw();
    link9.draw();
    link10.draw();
    link11.draw();
```

```
    link12.draw();
    link13.draw();
    link14.draw();
    link15.draw();
    glPopMatrix();   // come back to the upper torso
    link16.draw();   // start drawing the right side from the shoulder
    link17.draw();
    link18.draw();
    link19.draw();
    link20.draw();
    link21.draw();
    link22.draw();
    link23.draw();
    link24.draw();
}


//-------------------------------------------------------
// Function: reset()
// Purpose: reset all joint angles to their zero position
//-------------------------------------------------------
void Upperbody::reset()
{
    link1.reset();   // reset the joint angle in each link
    link2.reset();
    link3.reset();
    link4.reset();
    link5.reset();
    link6.reset();
    link7.reset();
    link8.reset();
    link9.reset();
    link10.reset();
    link11.reset();
    link12.reset();
    link13.reset();
    link14.reset();
    link15.reset();
    link16.reset();
    link17.reset();
    link18.reset();
    link19.reset();
    link20.reset();
    link21.reset();
    link22.reset();
    link23.reset();
    link24.reset();
}
```

```
//---------------------------------------------------------
// Function: set_link_length(int link, float length)
// Purpose: set the link length of a specified link
// Returns: TRUE if successful
//---------------------------------------------------------
int Upperbody::set_link_length(int link, float length)
{
   int success = 0;

   switch (link) {

      case 1:
         success = link1.set_draw_length(length);
         if (success){
            success = link2.set_inboard_link_length(length);
         }
         break;
      case 2:
         success = link2.set_draw_length(length);
         if (success){
            success = link3.set_inboard_link_length(length);
         }
         break;
      case 3:
         success = link3.set_draw_length(length);
         if (success){
            success = link4.set_inboard_link_length(length);
         }
         break;
      case 4:
         success = link4.set_draw_length(length);
         if (success){
            success = link5.set_inboard_link_length(length);
         }
         break;
      case 5:
         success = link5.set_draw_length(length);
         if (success){
            success = link6.set_inboard_link_length(length);
         }
         break;
      case 6:
         success = link6.set_draw_width(2*length);
         if (success){
            success = link7.set_inboard_link_length(length);
            success = link16.set_inboard_link_length(-length);
         }
         break;
```

```
      case 7:
         success = link7.set_draw_length(length);
         if (success){
            success = link8.set_inboard_link_length(length);
         }
         break;
      case 8:
         success = link8.set_draw_length(length);
         if (success){
            success = link9.set_inboard_link_length(length);
         }
         break;
      case 9:
         success = link9.set_draw_length(length);
         if (success){
            success = link10.set_inboard_link_length(length);
         }
         break;
      case 10:
         success = link10.set_draw_length(length);
         if (success){
            success = link11.set_inboard_link_length(length);
         }
         break;
      case 11:
         success = link11.set_draw_length(length);
         if (success){
            success = link12.set_inboard_link_length(length);
         }
         break;
      case 12:
         success = link12.set_draw_length(length);
         if (success){
            success = link13.set_inboard_link_length(length);
         }
         break;
      case 13:
         success = link13.set_draw_length(length);
         if (success){
            success = link14.set_inboard_link_length(length);
         }
         break;
      case 14:
         success = link14.set_draw_length(length);
         if (success){
            success = link15.set_inboard_link_length(length);
         }
         break;
      case 15:
         success = link15.set_draw_length(length);
         break;
```

```
      case 16:
         success = link16.set_draw_length(length);
         if (success){
            success = link17.set_inboard_link_length(length);
         }
         break;
      case 17:
         success = link17.set_draw_length(length);
         if (success){
            success = link18.set_inboard_link_length(length);
         }
         break;
      case 18:
         success = link18.set_draw_length(length);
         if (success){
            success = link19.set_inboard_link_length(length);
         }
         break;
      case 19:
         success = link19.set_draw_length(length);
         if (success){
            success = link20.set_inboard_link_length(length);
         }
         break;
      case 20:
         success = link20.set_draw_length(length);
         if (success){
            success = link21.set_inboard_link_length(length);
         }
         break;
      case 21:
         success = link21.set_draw_length(length);
         if (success){
            success = link22.set_inboard_link_length(length);
         }
         break;
      case 22:
         success = link22.set_draw_length(length);
         if (success){
            success = link23.set_inboard_link_length(length);
         }
         break;
      case 23:
         success = link23.set_draw_length(length);
         if (success){
            success = link24.set_inboard_link_length(length);
         }
         break;
      case 24:
         success = link24.set_draw_length(length);
         break;
```

```
      default:
         break;

   }
   return success;
}

//--------------------------------------------------------
// Function: set_joint_displacement(int link, float displacement)
// Purpose: set the joint displacement of a specified link
// Returns: TRUE if successful
//--------------------------------------------------------
int Upperbody::set_joint_displacement(int link, float displacement)
{
   int success = 0;

   switch (link) {

      case 1:
         success = link1.set_joint_displacement(displacement);
         break;
      case 2:
         success = link2.set_joint_displacement(displacement);
         break;
      case 3:
         success = link3.set_joint_displacement(displacement);
         break;
      case 4:
         success = link4.set_joint_displacement(displacement);
         break;
      case 5:
         success = link5.set_joint_displacement(displacement);
         break;
      case 6:
         success = link6.set_joint_displacement(displacement);

         break;
      case 7:
         success = link7.set_joint_displacement(displacement);
         if (success){
            success = link16.set_joint_displacement(displacement);
            success = link6.set_draw_length(displacement/
                           link6.get_draw_offset());
         }
         break;
      case 8:
         success = link8.set_joint_displacement(displacement);
         break;
      case 9:
         success = link9.set_joint_displacement(displacement);
         break;
```

```
      case 10:
         success = link10.set_joint_displacement(displacement);
         break;
      case 11:
         success = link11.set_joint_displacement(displacement);
         break;
      case 12:
         success = link12.set_joint_displacement(displacement);
         break;
      case 13:
         success = link13.set_joint_displacement(displacement);
         break;
      case 14:
         success = link14.set_joint_displacement(displacement);
         break;
      case 15:
         success = link15.set_joint_displacement(displacement);
         break;
      case 16:
         success = link16.set_joint_displacement(displacement);
         if (success){
            success = link7.set_joint_displacement(displacement);
            success = link6.set_draw_length(displacement/0.8);
         }
         break;
      case 17:
         success = link17.set_joint_displacement(displacement);
         break;
      case 18:
         success = link18.set_joint_displacement(displacement);
         break;
      case 19:
         success = link19.set_joint_displacement(displacement);
         break;
      case 20:
         success = link20.set_joint_displacement(displacement);
         break;
      case 21:
         success = link21.set_joint_displacement(displacement);
         break;
      case 22:
         success = link22.set_joint_displacement(displacement);
         break;
      case 23:
         success = link23.set_joint_displacement(displacement);
         break;
      case 24:
         success = link24.set_joint_displacement(displacement);
         break;
      default:
         break;
   }
   return success;
}
```

```
// ******************************************************
// FILENAME:  lowerbody.h
// PURPOSE:  declarations for the Lowerbody class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: definition of the Lowerbody class
// ******************************************************

#ifndef LOWERBODY_H
#define LOWERBODY_H

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>
#include <math.h>


class Lowerbody
{
   private:
      void drawDiamond(float, float, float, float, float,
              float, float);
      void computeNormal(const float* a, const float* b, const float* c,
                         float *result);



   public:
      Lowerbody(){};

//-------------------------------------------------------
// Function: draw()
// Purpose: draw the lowerbody
//        : the lowerbody is static and does not move
//-------------------------------------------------------
      void draw(){
         glRotated(-90.0, 0.0, 0.0, 1.0);
         drawDiamond(0.0, 0.0, 0.0, 12.0, 13.0, 6.5, 0.8);
         glRotated(30.0, 0.0, 0.0, 1.0);
         drawDiamond(10.5, 0.0, 0.0, 25.5, 9.5, 9.5, 0.2);
         drawDiamond(36.0, 0.0, 0.0, 27.5, 8.5, 8.5, 0.2);
         glRotated(-60.0, 0.0, 0.0, 1.0);
         drawDiamond(10.5, 0.0, 0.0, 25.5, 9.5, 9.5, 0.2);
         drawDiamond(36.0, 0.0, 0.0, 27.5, 8.5, 8.5, 0.2);
         glRotated(120.0, 0.0, 0.0, 1.0);
      }

};


#endif
```

```
// ******************************************************
// FILENAME:  lowerbody.cc
// PURPOSE:  function definition for the lowerbody class
//
// AUTHOR:  P F Skopowski
// DATE: 26 Nov 95
// COMMENTS: drawDiamond function
// ******************************************************

#include “lowerbody.h”


//-------------------------------------------------------
// Function: drawDiamond(float x, float y, float z,
//         : float length,float width, float depth,
//         : float offset)
// Purpose: draw a diamond with end at (x,y,z)
//        : use specified parameters
//        : draws diamond along the x-axis
//        : center of diamond can be offset
//-------------------------------------------------------
void Lowerbody::drawDiamond(float x, float y, float z,
              float length,
              float width,
              float depth,
              float offset)

// float x,y,z;        end of the diamond in 3-space.


{
   float midpoint;  // x coordinate for waist vertices of the diamond
   float halfwidth; // half the width
   float halfdepth; // half the depth

   float p[6][3];   // array to hold coords for the diamond vertices.
   float n[3];      // array to hold the normal vector


   // Compute the x-axis midpoint.
   midpoint=length*offset;

   // Compute half the dimensions.
   halfwidth = width/2.0;
   halfdepth = depth/2.0;


   // vertices.
   p[0][0]=x;
   p[0][1]=y;
   p[0][2]=z;
```

```
   p[1][0]=x+midpoint;
   p[1][1]=y;
   p[1][2]=z+halfdepth;

   p[2][0]=x+midpoint;
   p[2][1]=y+halfwidth;
   p[2][2]=z;

   p[3][0]=x+midpoint;
   p[3][1]=y;
   p[3][2]=z-halfdepth;

   p[4][0]=x+midpoint;
   p[4][1]=y-halfwidth;
   p[4][2]=z;

   p[5][0]=x+length;
   p[5][1]=y;
   p[5][2]=z;

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.


   // Compute and set normal for the first side
   computeNormal(p[0], p[1], p[2], n);
   glNormal3fv(n);

   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[1]);
      glVertex3fv(p[2]);
   glEnd();

   //glColor3f(0.0, 0.0, 0.0);   // Set the color to black.

   // Compute and set normal for the first side
   computeNormal(p[0], p[2], p[3], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[2]);
      glVertex3fv(p[3]);
   glEnd();

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.

   // Compute and set normal for the first side
   computeNormal(p[0], p[3], p[4], n);
   glNormal3fv(n);
```

```
   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[3]);
      glVertex3fv(p[4]);
   glEnd();

   //glColor3f(0.5, 0.5, 0.5);   // Set the color to gray.

   // Compute and set normal for the first side
   computeNormal(p[0], p[4], p[1], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[0]);
      glVertex3fv(p[4]);
      glVertex3fv(p[1]);
   glEnd();

   //glColor3f(0.5, 0.5, 0.5);   // Set the color to gray.

   // Compute and set normal for the first side
   computeNormal(p[5], p[2], p[1], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[2]);
      glVertex3fv(p[1]);
   glEnd();

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.

   // Compute and set normal for the first side
   computeNormal(p[5], p[3], p[2], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[3]);
      glVertex3fv(p[2]);
   glEnd();

   //glColor3f(0.0, 0.0, 0.0);   // Set the color to black.

   // Compute and set normal for the first side
   computeNormal(p[5], p[4], p[3], n);
   glNormal3fv(n);
```

```
   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[4]);
      glVertex3fv(p[3]);
   glEnd();

   //glColor3f(1.0, 1.0, 1.0);   // Set the color to white.

   // Compute and set normal for the first side
   computeNormal(p[5], p[1], p[4], n);
   glNormal3fv(n);


   glBegin(GL_POLYGON);
      glVertex3fv(p[5]);
      glVertex3fv(p[1]);
      glVertex3fv(p[4]);
   glEnd();

   // the diamond is drawn.

}




//-------------------------------------------------------------
// Function: computeNormal(const float* a, const float* b,
//         : const float* c, float *result)
// Purpose: compute normal vector to a triangular polygon
//-------------------------------------------------------------
void Lowerbody::computeNormal(const float* a, const float* b, const float* c,
                          float *result)

{
   float x[3];
   float y[3];
   float magnitude;

   // compute the first vector

   x[0] = b[0] - a[0];
   x[1] = b[1] - a[1];
   x[2] = b[2] - a[2];

   // compute the second vector

   y[0] = c[0] - a[0];
   y[1] = c[1] - a[1];
   y[2] = c[2] - a[2];
```

```
   // compute the cross product vector

   result[0] = x[1] * y[2] - x[2] * y[1];
   result[1] = x[2] * y[0] - x[0] * y[2];
   result[2] = x[0] * y[1] - x[1] * y[0];

   //normalize the result

   magnitude = sqrt(result[0] * result[0] + result[1] * result[1] +
                    result[2] * result[2]);

   result[0] = result[0]/magnitude;
   result[1] = result[1]/magnitude;
   result[2] = result[2]/magnitude;


}
```

```
// ******************************************************
// FILENAME:  body.h
// PURPOSE:  declarations for the Body class
//        :  uses angles-only tracking technique
// AUTHOR:  P F Skopowski
// DATE: 1 Mar 96
// COMMENTS: definition of the Body class
// ******************************************************

#ifndef BODY_H
#define BODY_H

#define PF_CPLUSPLUS_API 0
#include <Performer/pf.h>
#include “upperbody.h”
#include “lowerbody.h”
#include “FastrakClass.h”



class Body
{
   private:
      Upperbody upperbody;
      Lowerbody lowerbody;

      int valid;

      FastrakClass *fastrak_unit;

      FSTK_stations torso_sensor;
      FSTK_stations upperarm_sensor;
      FSTK_stations lowerarm_sensor;
      FSTK_stations hand_sensor;

      // Fastrak related coordinate systems
      pfMatrix H_tx_to_ts, H_tx_to_uas, H_tx_to_las, H_tx_to_hs;

      // Calibration matrices
      pfMatrix H_ts_to_link0, H_uas_to_link20;
      pfMatrix H_las_to_link21, H_hs_to_link24;
      pfMatrix H_ts_to_screen;
      pfMatrix H_ts_to_posit0;
      float x_offset, y_offset, z_offset;
      float x_posit, y_posit, z_posit;
      float x_ref, y_ref, z_ref;

      // Graphical model related coordinate systems
      pfMatrix H_screen, H0, H20, H21, H24;
```

```
      // Body part lengths
      float spine_shoulder_length, uarm_length, larm_length, hand_length;

      void outputHMatrix(pfMatrix H_mat);


   public:
      Body(const char *cfg_filename);

      ~Body();

      void rotate (double *);

      void rotate_increment (double *);

      void draw();

      void reset();

      int  exists() { return valid; }

      void get_all_inputs();

      int  calibrate();

      int  set_joint_angles();

      int calculate_joint_angles(double *);

      int set_link_length(int, float);

      int set_joint_displacement(int, float);

};


#endif
```

```
// *******************************************************
// FILENAME:  body.cc
// PURPOSE:  functions for the Body class
//        :  angles-only tracking technique
// AUTHOR:  P F Skopowski
// DATE: 1 Mar 96
// COMMENTS: functions for the Body class
// *******************************************************

#include <math.h>
#include <iostream.h>
#include "body.h"


//-------------------------------------------------------
// Function: Body(const char *config_filename)
// Purpose: constructor of the body type
//        : creates and initializes FastrakClass object
//        : uses fastrak.dat configuration file
// Returns: body class object
//-------------------------------------------------------
Body::Body(const char *config_filename)
{
   valid = FALSE;

   fastrak_unit = NULL;

   // open configuration file
   ifstream config_fileobj(config_filename);
   if (!config_fileobj) {
      cerr << "Error: opening configuration file: "
           << config_filename << endl;
      return;
   }


   // initialize matrices & variables
   pfMakeIdentMat(H_tx_to_ts);
   pfMakeIdentMat(H_tx_to_uas);
   pfMakeIdentMat(H_tx_to_las);
   pfMakeIdentMat(H_tx_to_hs);
   pfMakeIdentMat(H_ts_to_screen);
   pfMakeIdentMat(H_ts_to_link0);
   pfMakeIdentMat(H_ts_to_posit0);
   pfMakeIdentMat(H_uas_to_link20);
   pfMakeIdentMat(H_las_to_link21);
   pfMakeIdentMat(H_hs_to_link24);
   pfMakeIdentMat(H_screen);
   pfMakeIdentMat(H0);
   pfMakeIdentMat(H20);
   pfMakeIdentMat(H21);
   pfMakeIdentMat(H24);
   x_offset = 0.0;
```

```
   y_offset = 0.0;
   z_offset = 0.0;


   //initialize Fastrak
   fastrak_unit = new FastrakClass(config_fileobj);

   if (fastrak_unit->exists()) {
      if (fastrak_unit->getState(FSTK_STATION1))
         torso_sensor = FSTK_STATION1;
      if (fastrak_unit->getState(FSTK_STATION2))
         upperarm_sensor  = FSTK_STATION2;
      if (fastrak_unit->getState(FSTK_STATION3))
         lowerarm_sensor = FSTK_STATION3;
      if (fastrak_unit->getState(FSTK_STATION4))
         hand_sensor = FSTK_STATION4;

      valid = TRUE;
   }

}


//-------------------------------------------------------
// Function: ~Body()
// Purpose: destructor of the body type
//-------------------------------------------------------
Body::~Body()
{
   if ((fastrak_unit != NULL) && (fastrak_unit->exists())) {
      delete fastrak_unit;
      fastrak_unit = NULL;
   }
}


//-------------------------------------------------------
// Function: rotate (double *angles)
// Purpose: set upperbody joint angles
//        : uses the passed in array of values
//-------------------------------------------------------
void Body::rotate (double *angles)
{
    upperbody.rotate(angles);
}
```

```
//---------------------------------------------------------
// Function: rotate_increment (double *increment_angles)
// Purpose: increment upperbody joint angles
//        : uses the passed in array of values
//---------------------------------------------------------
void Body::rotate_increment (double *increment_angles)
{
    upperbody.rotate_increment(increment_angles);
}


//---------------------------------------------------------
// Function: draw()
// Purpose: draw the body in the proper position
//---------------------------------------------------------
void Body::draw()
{
    // determine where to start drawing the upperbody
    pfMultMat(H_screen, H_tx_to_ts, H_ts_to_screen);
    pfMatrix temp;
    pfMultMat(temp, H_tx_to_ts, H_ts_to_posit0);
    x_offset = temp[0][3] - H_tx_to_ts[0][3];
    y_offset = temp[1][3] - H_tx_to_ts[1][3];
    z_offset = temp[2][3] - H_tx_to_ts[2][3];
    x_posit = H_tx_to_ts[0][3] - x_ref + x_offset;
    y_posit = H_tx_to_ts[1][3] - y_ref + y_offset;
    z_posit = H_tx_to_ts[2][3] - z_ref + z_offset;

    // set the openGL matrix (ie. array)
    GLfloat H_body[16];
    H_body[0] = H_screen[0][0];
    H_body[1] = H_screen[1][0];
    H_body[2] = H_screen[2][0];
    H_body[3] = H_screen[3][0];
    H_body[4] = H_screen[0][1];
    H_body[5] = H_screen[1][1];
    H_body[6] = H_screen[2][1];
    H_body[7] = H_screen[3][1];
    H_body[8] = H_screen[0][2];
    H_body[9] = H_screen[1][2];
    H_body[10] = H_screen[2][2];
    H_body[11] = H_screen[3][2];
    H_body[12] = x_posit;
    H_body[13] = y_posit;
    H_body[14] = z_posit;
    H_body[15] = H_screen[3][3];

    glMatrixMode(GL_MODELVIEW);


    glPushMatrix();

    // align the tx and screen coord systems
```

```
    glRotated(90.0, 1.0, 0.0, 0.0);
    glRotated(-90.0, 0.0, 0.0, 1.0);

    glMultMatrixf(H_body);

    upperbody.draw();

    glPopMatrix();

    //  lowerbody.draw();
}


//-------------------------------------------------------
// Function: reset()
// Purpose: reset upperbody joint angles
//-------------------------------------------------------
void Body::reset()
{
    upperbody.reset();
}


//-------------------------------------------------------
// Function: set_link_length(int link, float length)
// Purpose: set  a  specified link's length
//        : used to size the link to the user
// Returns: TRUE if successful
//-------------------------------------------------------
int Body::set_link_length(int link, float length)
{
    if(upperbody.set_link_length(link, length)){
       return TRUE;
    }
    return FALSE;
}


//-------------------------------------------------------
// Function: set_joint_displacement(int link, float length)
// Purpose: set  a  specified link's joint displacement
//        : used to size the link to the user
// Returns: TRUE if successful
//-------------------------------------------------------
int Body::set_joint_displacement(int link, float length)
{
    if(upperbody.set_joint_displacement(link, length)){
       return TRUE;
    }
    return FALSE;
}
```

```
//-------------------------------------------------------
// Function: get_all_inputs()
// Purpose: get inputs from the fastrak trackers
//        : called to copy latest sample from second buffer
//        : implemented for double buffering to reduce
//        : lock overhead
//        : called once at the beginning of each frame
// Comment: original interface design by Scott McMillan
//-------------------------------------------------------
void Body::get_all_inputs()
{

   if (fastrak_unit->exists()) {
      fastrak_unit->copyBuffer();

      fastrak_unit->getHMatrix(torso_sensor, H_tx_to_ts);
      fastrak_unit->getHMatrix(upperarm_sensor, H_tx_to_uas);
      fastrak_unit->getHMatrix(lowerarm_sensor,  H_tx_to_las);
      fastrak_unit->getHMatrix(hand_sensor, H_tx_to_hs);

   }
}


//-------------------------------------------------------
// Function: output
// Purpose: output homogeneous transformation matrix (4x4)
//-------------------------------------------------------
void Body::outputHMatrix(pfMatrix Hmat)
{
   for (int i=0; i<4; i++)
      printf(" %6.3f %6.3f %6.3f %6.3f\n",
             Hmat[i][0], Hmat[i][1], Hmat[i][2], Hmat[i][3]);
   printf("\n");
}


//-------------------------------------------------------
// Function: set_joint_angles()
// Purpose: Set the body's joint angles using fastrak data
// Returns: TRUE if successful
//-------------------------------------------------------
int Body::set_joint_angles()
{
   int valid = FALSE;

   double angles[25];

   for (int i = 0; i < 25; i++){
      angles[i] = 0.0;
   }

   valid = calculate_joint_angles(angles);
```

```
   if (valid){
      rotate(angles);
   }

   return valid;
}


//-------------------------------------------------------
// Function: calculate_joint_angles(double *)
// Purpose: calculate inverse kinematics
//        : return the joint angles
//        : get_all_inputs must run first to update data
// Returns: TRUE if successful
//-------------------------------------------------------
int Body::calculate_joint_angles(double *angles)
{
   int valid = FALSE;

   double theta18 = 0.0;
   double theta19 = 0.0;
   double theta20 = 0.0;
   double theta21 = 0.0;
   double theta22 = 0.0;
   double theta23 = 0.0;
   double theta24 = 0.0;


   const double deg_to_rad = .017453292519943295;

   if (fastrak_unit->exists()) {
      // must call get_all_inputs() first
      valid = TRUE;

      // convert reported data using calibration matrices
      pfMultMat(H0, H_tx_to_ts, H_ts_to_link0);
      pfMultMat(H20, H_tx_to_uas, H_uas_to_link20);
      pfMultMat(H21, H_tx_to_las, H_las_to_link21);
      pfMultMat(H24, H_tx_to_hs, H_hs_to_link24);

      // compute T_17_to_20
      pfMatrix T_17_to_20, H_temp;
      pfMatrix T_17_to_0 = {{ 0.0,  1.0,  0.0,  7.5},
                            { 0.0,  0.0, -1.0, 40.0 },
                            {-1.0,  0.0,  0.0,  0.0 },
                            { 0.0,  0.0,  0.0,  1.0 }};

      pfMatrix H0_inv;
      pfInvertFullMat(H0_inv, H0);

      pfMultMat(H_temp, H0_inv, H20);
      pfMultMat(T_17_to_20, T_17_to_0, H_temp);
```

```
      // get the data from T_17_to_20
      double a2 = T_17_to_20[1][0];
      double b2 = T_17_to_20[1][1];
      double c3 = T_17_to_20[2][2];
      double c2 = T_17_to_20[1][2];
      double c1 = T_17_to_20[0][2];

      // compute the sin of theta19
      double sin_theta19 = sqrt(a2 * a2 + b2 * b2);

      // check for zero
      if (sin_theta19 < 0.001){
         sin_theta19 = 0.001;
      }

      // set the sign of the answer
      if (c3 < 0.0){
         sin_theta19 *= -1.0;
      }

      // compute the angles
      theta19 = atan2(sin_theta19, c2);
      theta20 = atan2(b2/sin_theta19, -a2/sin_theta19);
      theta18 = atan2(c3/sin_theta19, c1/sin_theta19);


      // compute T_20_to_21
      pfMatrix T_20_to_21, H20_inv;

      pfInvertFullMat(H20_inv, H20);

      pfMultMat(T_20_to_21, H20_inv, H21);

      // get the data from T_20_to_21
      float a3 = T_20_to_21[2][0];
      float b3 = T_20_to_21[2][1];

      // compute the angle
      theta21 = atan2(a3, b3);

      pfMatrix T_21_to_24, H21_inv;

      pfInvertFullMat(H21_inv, H21);

      pfMultMat(T_21_to_24, H21_inv, H24);

      // get the data from H24
      a3 = T_21_to_24[2][0];
      b3 = T_21_to_24[2][1];
      c3 = T_21_to_24[2][2];
      c2 = T_21_to_24[1][2];
      c1 = T_21_to_24[0][2];
```

```
      // compute the sin of theta23
      double sin_theta23 = -sqrt(a3 * a3 + b3 * b3);

      // check for zero
      if (sin_theta23 > -0.001){
         sin_theta23 = -0.001;
      }


      // compute the angles
      theta23 = atan2(sin_theta23, -c3);
      theta24 = atan2(b3/sin_theta23, -a3/sin_theta23);
      theta22 = atan2(-c2/sin_theta23, -c1/sin_theta23);

      // convert all angles to degrees

      theta18 /= deg_to_rad;
      theta19 /= deg_to_rad;
      theta20 /= deg_to_rad;
      theta21 /= deg_to_rad;
      theta22 /= deg_to_rad;
      theta23 /= deg_to_rad;
      theta24 /= deg_to_rad;

      angles[1]  =    0.0;
      angles[2]  =  -90.0;
      angles[3]  =    0.0;
      angles[4]  =   90.0;
      angles[5]  =   90.0;
      angles[6]  =  180.0;
      angles[7]  =    0.0;
      angles[8]  =    0.0;
      angles[9]  =   90.0;
      angles[10] =   90.0;
      angles[11] =    0.0;
      angles[12] =    0.0;
      angles[13] =    0.0;
      angles[14] =  -90.0;
      angles[15] =    0.0;
      angles[16] =  180.0;
      angles[17] =    0.0;
      angles[18] =  theta18;
      angles[19] =  theta19;
      angles[20] =  theta20;
      angles[21] =  theta21;
      angles[22] =  theta22;
      angles[23] =  theta23;
      angles[24] =  theta24;
   }

   return valid;
}
```

```
//---------------------------------------------------------
// Function: calibrate()
// Purpose: size the upperbody model to the user
//        : calibrate the trackers
// Returns: TRUE if successful
//---------------------------------------------------------
int Body::calibrate()
{
   int valid = FALSE;

   pfMatrix H_torso_reported, H_uarm_reported;
   pfMatrix H_larm_reported, H_hand_reported;

   pfMakeIdentMat(H_torso_reported);
   pfMakeIdentMat(H_uarm_reported);
   pfMakeIdentMat(H_larm_reported);
   pfMakeIdentMat(H_hand_reported);

   if (fastrak_unit->exists()) {
      valid = TRUE;
      char str;


      // request upperbody dimensions
      cerr << endl << "Input spine to shoulder length (cm): ";
      cin >> spine_shoulder_length;
      cerr << "Input upper arm length: ";
      cin >> uarm_length;
      cerr << "Input lower arm length: ";
      cin >> larm_length;
      cerr << "Input hand length: ";
      cin >> hand_length;
      cin.get(str);



      // set upperbody dimensions to that of the user
      set_link_length(3, 21.0);
      set_link_length(6, 0.36 * spine_shoulder_length);
      set_joint_displacement(16, 26.0);
      set_link_length(17, 0.64 * spine_shoulder_length);
      set_link_length(8, 0.64 * spine_shoulder_length);
      set_link_length(20, uarm_length);
      set_link_length(11, uarm_length);
      set_link_length(21, larm_length);
      set_link_length(12, larm_length);
      set_link_length(24, hand_length);
      set_link_length(15, hand_length);
```

```
cerr << endl << "Calibrating sensor orientation in 3 seconds..." << endl;
cerr << "Press <Enter> to start count-down: ";
cin.get(str);
for (int i=0; i<3; i++) {
   sleep(1);
   cerr << (char) 7;
}

// this code allows the fastrak to do the calibration for torso
//float angles[3] = {90.0, -90.0, 180.0};
//fastrak_unit->setBoresight(torso_sensor, angles);

// get the data to compute the calibration matrices
fastrak_unit->copyBuffer();
fastrak_unit->getHMatrix(torso_sensor, H_torso_reported);
fastrak_unit->getHMatrix(upperarm_sensor, H_uarm_reported);
fastrak_unit->getHMatrix(lowerarm_sensor, H_larm_reported);
fastrak_unit->getHMatrix(hand_sensor,  H_hand_reported);



// compute the calibration matrices

// compute torso sensor calibration matrix
pfMatrix H_torso_reported_inv;

pfMatrix H_ts_desired = {{ 1.0,  0.0,  0.0, 0.0},
                         { 0.0,  1.0,  0.0, 0.0},
                         { 0.0,  0.0,  1.0, 0.0},
                         { 0.0,  0.0,  0.0, 1.0}};

pfInvertFullMat(H_torso_reported_inv, H_torso_reported);

pfMultMat(H_ts_to_link0, H_torso_reported_inv, H_ts_desired);

pfMatrix H_ts_desired2 = {{ 1.0,  0.0,  0.0, 0.0},
                          { 0.0,  1.0,  0.0, 0.0},
                          { 0.0,  0.0,  1.0, 0.0},
                          { 0.0,  0.0,  0.0, 1.0}};

pfMultMat(H_ts_to_screen, H_torso_reported_inv, H_ts_desired2);
pfSetMatCol(H_ts_to_screen, 3, 0.0, 0.0, 0.0, 1.0);


// compute torso sensor calibration matrix for posit0 tracking
// some necessary matrices
pfMatrix R_tx_to_ts, R_ts_to_tx;

pfCopyMat(R_tx_to_ts, H_torso_reported);

// set posit col to zero to work with rotation matrix only
pfSetMatCol(R_tx_to_ts, 3, 0.0, 0.0, 0.0, 1.0);
```

```
    // get the inverse rotation matrix
    pfTransposeMat(R_ts_to_tx, R_tx_to_ts);

    // determine suitable offsets from ts and link0
    float x_offset = 0.0;
    float y_offset = 0.0;
    float z_offset = 47.0;

    // the offset from ts to link0 in world coordinates
    pfMatrix P_offset_ts_to_link0  = {{ 1.0,  0.0,  0.0,   x_offset},
                                     { 0.0,  1.0,  0.0,   y_offset},
                                     { 0.0,  0.0,  1.0,   z_offset},
                                     { 0.0,  0.0,  0.0,   1.0}};
    pfMatrix temp;
    pfMultMat(temp, R_ts_to_tx, P_offset_ts_to_link0);
    pfSetMatCol(H_ts_to_posit0, 3, temp[0][3], temp[1][3], temp[2][3], 1.0);
    x_ref = H_torso_reported[0][3];
    y_ref = H_torso_reported[1][3];
    z_ref = H_torso_reported[2][3];


    // compute upper arm sensor calibration matrix
    pfMatrix H_uarm_reported_inv;

    pfMatrix H_uarm_desired = {{ 0.0,  0.0, -1.0, 0.0},
                               { 0.0,  1.0,  0.0, 0.0},
                               { 1.0,  0.0,  0.0, 0.0},
                               { 0.0,  0.0,  0.0, 1.0}};

    pfInvertFullMat(H_uarm_reported_inv, H_uarm_reported);

    pfMultMat(H_uas_to_link20, H_uarm_reported_inv, H_uarm_desired);


    // compute lower arm sensor calibration matrix
    pfMatrix H_larm_reported_inv;
    // place elbow in 90 bend
    pfMatrix H_larm_desired = {{ 1.0,  0.0,  0.0, 0.0},
                               { 0.0,  0.0, -1.0, 0.0},
                               { 0.0,  1.0,  0.0, 0.0},
                               { 0.0,  0.0,  0.0, 1.0}};
    // was previously this, for straight down
    //pfMatrix H_larm_desired = {{ 0.0, -1.0,  0.0, 0.0},
    //                           { 0.0,  0.0, -1.0, 0.0},
    //                           { 1.0,  0.0,  0.0, 0.0},
    //                           { 0.0,  0.0,  0.0, 1.0}};

    pfInvertFullMat(H_larm_reported_inv, H_larm_reported);

    pfMultMat(H_las_to_link21, H_larm_reported_inv, H_larm_desired);

    // compute hand sensor calibration matrix
    pfMatrix H_hand_reported_inv;
```

```
      // place hand straight out with elbow bent
      pfMatrix H_hand_desired = {{ 0.0,  0.0,  1.0, 0.0},
                                 {-1.0,  0.0,  0.0, 0.0},
                                 { 0.0, -1.0,  0.0, 0.0},
                                 { 0.0,  0.0,  0.0, 1.0}};

      // was previously this, for straight down
      //pfMatrix H_hand_desired = {{ 0.0,  1.0,  0.0, 0.0},
      //                           {-1.0,  0.0,  0.0, 0.0},
      //                           { 0.0,  0.0,  1.0, 0.0},
      //                           { 0.0,  0.0,  0.0, 1.0}};


      pfInvertFullMat(H_hand_reported_inv, H_hand_reported);

      pfMultMat(H_hs_to_link24, H_hand_reported_inv, H_hand_desired);

   }

   return valid;
}
```

APPENDIX B: FASTRAK DEVICE DRIVER


```
// ********************************************************************
// File    : FastrakClass.h
// Author  : Scott McMillan
//         : Naval Postgraduate School
//         : Code CSMs
//         : Monterey, CA 93943
//         : mcmillan@cs.nps.navy.mil
// Project : NPSNET - Individual Combatants/Insertion of Humans into VEs
// Created : August 1995
// Summary : This file contains the declarations for a C++ class to
//         : manage the Polhemus 3Space Fastrak.
//         :
//         : For detailed information on the operation of the Fastrak,
//         : refer to the 3SPACE USER'S MANUAL.
//         :
//         : This program was based on the ISOTRACK program written by
//         : Paul T. Barham in Sept. 1993 for single sensor case.  Major
//         : modifications have been made to adapt to multiple sensor
//         : case.  The resulting code, written by Jiang Zhu in
//         : Jan. 1995, underwent another major modification to support
//         : binary data in continuous mode.
// ********************************************************************

/*
 * Copyright (c) 1995,
 *    Naval Postgraduate School
 *    Computer Graphics and Video Laboratory
 *    NPSNET Research Group
 *    npsnet@cs.nps.navy.mil
 *
 *
 *    Permission to use, copy, and modify this software and its
 * documentation for any non-commercial purpose is hereby granted
 * without fee, provided that (i) the above copyright notices and the
 * following permission notices appear in ALL copies of the software
 * and related documentation, and (ii) The Naval Postgraduate School
 * Computer Graphics and Video Laboratory and the NPSNET Research Group
 * be given written credit in your software's written documentation and
 * be given graphical credit on any start-up/credit screen your
 * software generates.
 *    This restriction helps justify our research efforts to the
 * sponsors who fund our research.
 *
 *    Do not redistribute this code without the express written
 * consent of the NPSNET Research Group. (E-mail communication and our
 * confirmation qualifies as written permission.) As stated above, this
 * restriction helps justify our research efforts to the sponsors who
 * fund our research.
 *
 *    This software was designed and implemented at U.S. Government
```

```
 * expense and by employees of the U.S. Government.  It is illegal
 * to charge any U.S. Government agency for its partial or full use.
 *
 * THE SOFTWARE IS PROVIDED “AS IS” AND WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
 * WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
 *
 * E-Mail addresses:
 *    npsnet@cs.nps.navy.mil
 *       General code questions, concerns, comments, requests for
 *       distributions and documentation, and bug reports.
 *    npsnet-info@cs.nps.navy.mil
 *       Contact principle investigators.
 *       Overall research project information and funding.
 *       Requests for demonstations.
 *
 * Thank you to our sponsors:  ARL, STRICOM, TRAC, ARPA and DMSO.
 */

#ifndef __NPS_FASTRAK_CLASS__
#define __NPS_FASTRAK_CLASS__

#include <ulocks.h>
#include <fstream.h>

// Boolean values
#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

const int PORT_NAME_SIZE = 48;

// Assumptions and limitations which influence the use of the FASTRAK
// program:
//
// 1). I assume that the user of the FASTRAK program will use it in a
//     single process, say the application process in PERFORMER, that
//     is, a instance of the “FastrakClass” class will only be used in a
//     process in which it is created.  Hopefully, this is not too
//     restricted.
//
//     The problem with the current version when it is used in multiple
//     processes is that the read_data(), read_posorient() and
//     read_homomatrix() methods are not guarded with the
//     data-record-parameter-updating binary semophere.  Instead, a lock
//     is used for guarding data buffer switching. Refer to the
//     implementation for details.
//
//     It is easy to fix this problem. Basically, what you need to do is
```

```
//     to guard the two methods with the above binary semophere. The
//     cost is that now the
//     read_data()/read_posorient()/read_homomatrix() method and the
//     get_packet() method are almost totally mutually exclusive, which
//     may slow the system performance down. If this is done, the lock
//     for guarding data buffer switching can be eliminated.
//
// 2). The FASTRAK program was witten so that it can be used to process
//     any number of sensors. The only thing you need to do is to change
//     the constant FSTK_NUM_STATIONS to the number of sensors you
//     have. However, there is a limit caused by the system on the
//     number of sensors you can use. Basically, the problem is that the
//     data buffer size, BUFFER_SIZE, is constrained by the size of the
//     largest one-dimensional array allowed by the system.


// Conventions used in this file and FastrakClass.cc:
//
//    All the constants and data types intended to be used by the user
//    of the FSTK start with the prefix FSTK. Those that do not have
//    this prefix should be used only in this file and FastrakClass.cc.
//
//    All constants are in capital letters.

// Terms used in this file and FastrakClass.cc:
//
//    station: Each trasmitter and receiver pair is called a station in
//             the 3SPACE USER'S MANUAL.

// The algorithm:
//
//    In single process mode, a single process, which is the one which
//    creates the the "FastrakClass" object, requests a required type of data
//    packet from the FSTK when it needs one.
//
//    In multiprocess mode, the process which creates the "FastrakClass"
//    object spawns a light weight child process which continuously
//    polls the FSTK to get a required type of data packet, which is the
//    data producer and runs in parallel with the parent process, the
//    data consumer.
//
//    The data packet is decomposed in the parent (or single) process,
//    the consumer, to generate the required type of data when a data is
//    requested by the user of the FSTK.

// the masks used to specify the types of data requested from the FSTK
#define FSTK_COORD_MASK       0x001   // Cartesian coordinate (X,Y,Z)
#define FSTK_EULER_MASK       0x002   // Euler angles (Azim,Elev,Roll)
#define FSTK_XCOS_MASK        0x004   // X-axis directional cosines
#define FSTK_YCOS_MASK        0x008   // Y-axis directional cosines
#define FSTK_ZCOS_MASK        0x010   // Z-axis directional cosines
#define FSTK_QUAT_MASK        0x020   // Quaternion (W, X, Y, Z)
#define FSTK_16BIT_COORD_MASK 0x040   // 16BIT format coordinate data
```

```
#define FSTK_16BIT_EULER_MASK 0x080   // 16BIT format euler angles
#define FSTK_16BIT_QUAT_MASK  0x100   // 16BIT format quaternion
#define FSTK_CRLF_MASK        0x200   // the framing CR/LF characters
#define FSTK_DEFAULT_MASK     0x0c0   // 16BIT COORD and EULER_MASKs

// The sizes of the types of data returned from the FSTK -- in bytes
// IEEE single precision floating point format has 4 bytes per number
// 16BIT obviously has two bytes per number.
#define FSTK_HEADER_SIZE      3   // data record header from the FSTK
#define FSTK_COORD_SIZE       12  // Position coordinates
#define FSTK_EULER_SIZE       12  // Euler angles
#define FSTK_XCOS_SIZE        12  // X-axis directional cosines
#define FSTK_YCOS_SIZE        12  // Y-axis directional cosines
#define FSTK_ZCOS_SIZE        12  // Z-axis directional cosines
#define FSTK_QUAT_SIZE        16  // Quaternion
#define FSTK_16BIT_COORD_SIZE 6   // 16BIT format coordinates
#define FSTK_16BIT_EULER_SIZE 6   // 16BIT format euler angles
#define FSTK_16BIT_QUAT_SIZE  8   // 16BIT format quaternions
#define FSTK_CRLF_SIZE        2   // Carriage Return, Line Feed

// There can be up to 4 stations active at the same time in the FASTRAK.
#define FSTK_NUM_STATIONS 4

#define MAX_PACKET_SIZE   91  // All of the above summed together
#define BUFFER_SIZE       364 // MAX_PACKET_SIZE*FSTK_NUM_STATIONS

// 16BIT format requires scaling information for the various types of
// data:
#define FSTK_16BIT_TO_CM        (300.0/8192)
#define FSTK_16BIT_TO_INCHES    (118.11/8192)
#define FSTK_16BIT_TO_DEGREES   (180.0/8192)
#define FSTK_16BIT_TO_QUAT      (1.0/8192)

// Station numbers
enum FSTK_stations
      { FSTK_STATION1 = 0,    // This may be bad programming style, but
        FSTK_STATION2,        // these are used to index into arrays.
        FSTK_STATION3,
        FSTK_STATION4 };

// The FASTTRACk can return up to 6 different types of data.
#define FSTK_NUM_DATATYPES  9               // number of data types
enum FSTK_datatypes
        { FSTK_COORD_TYPE = 0, // This may be bad programming style, but
          FSTK_EULER_TYPE,     // these are used to index into arrays.
          FSTK_XCOS_TYPE,
          FSTK_YCOS_TYPE,
          FSTK_ZCOS_TYPE,
          FSTK_QUAT_TYPE,
          FSTK_16BIT_COORD_TYPE,
          FSTK_16BIT_EULER_TYPE,
          FSTK_16BIT_QUAT_TYPE };
```

```
// units used to measure the FSTK positions
enum FSTK_units
        { FSTK_INCH, FSTK_CENTIMETER };

//========================================================================
// FastrakClass definitions
//========================================================================

class FastrakClass
{
private:
   int   port_fd;                      // the serial port file descriptor
   char  port_name[PORT_NAME_SIZE];     // the name of the Fastrak port

   // info for the individual byte buffers for the four Fastrak stations.
   char  datarec[FSTK_NUM_STATIONS][MAX_PACKET_SIZE];
   char  datarec_buf[FSTK_NUM_STATIONS][MAX_PACKET_SIZE];
                                        // data records for each station
   short max_datarec_size;              // size of the largest station pkt
   short datarec_size[FSTK_NUM_STATIONS];  // data rec. size for each station
   short datatype_mask[FSTK_NUM_STATIONS]; // data types for each station
   short datatype_start[FSTK_NUM_STATIONS][FSTK_NUM_DATATYPES];
                                        // the position of each requested
                                        //   type in the data record

   short fstk_packet_size;           // the sum of the data record sizes
                                     //   for all the active stations, i.e.,
                                     //   the size of a complete data
                                     //   packet returned from the Fastrak
   char  read_buffer[BUFFER_SIZE];   // pollContinuously's temporary buffer
   unsigned int   read_index;        // current location in temp buffer

   // Process ID and process for the serial port polling process.
   int parpoll_pid;
   friend void pollContinuously(void *); // the sproc'ed fcn to read port
   void getPacket();                 // read a packet from the FSTK


   // locks and binary semaphores for ensuring mutual exclusive access
   // to critical sections.
   // Note that a boolean flag, param_setting, is used together with the
   // semaphore.  Basically, when the consumer process is already
   // holding the binary semaphore, it should not request the semaphore
   // again. Otherwise, deadlock would occur.  It is needed when a
   // consumer data record parameter setting method needs to call
   // another such method. Now there is one such case, set_state()
   // calling specify_datatype(). Note that param_set_flag should never
   // be used in the data producer process.
   ulock_t datalock;    // a lock used to guide switching data buffers
   usema_t *paramsema;  // a binary semaphore used to guide setting data
                        //    packet parameters.
   usptr_t *arena;      // arena used to create lock and semaphore
```

```
// Boolean flags
int param_set_flag;  // TRUE if the data packet parameters are being set.
int valid_flag;      // TRUE if initializing the FSTK is successfull
int is_polling_flag; // TRUE if the papallel polling process is not
                     //    being suspended
volatile int data_ready_flag; // TRUE if new data has been read
                              //    after control parameter update
volatile int kill_flag;       // TRUE if exiting the parallel polling
                              //    process has been requested

// Fastrak state variables:
int active_setting[FSTK_NUM_STATIONS];   // h/w switch settings
int active_state[FSTK_NUM_STATIONS];     // s/w setting
float alignment[FSTK_NUM_STATIONS][3][3];// pp. 42-50 in User's Manual
float boresight[FSTK_NUM_STATIONS][3];   // pp. 51-55
float hemisphere[FSTK_NUM_STATIONS][3];  // pp. 88-92
FSTK_units units;                        // CENTIMETERS and INCHES

// private methods
void initState();                    // init the member variables
int  readConfig(ifstream &config_fileobj); // read the config file
int  openIOPort();                   // open the FSTK serial io-port
int  initMultiprocessing();          // initialize multiprocess mode

int  checkState();                   // check which station is active

void prepareToRead();                // parallel/serial i/f to getPacket

int  sendCommand(char* command,      // send a command to the FSTK
                 int length,
                 char* source);

void convertData(char* data,         // convert IEEE buffer data to n
                 int num_floats,     // floats:  DOS ordered bytes to
                 float data_dest[]); // Unix (reversed)

void convert16BITData(char* data,    // convert a 16BIT format buffer
                 int num_floats,     // to n IEEE floating point nums
                 float scale,
                 float data_dest[]);


// functions for debugging and error checking
void debugData(char *data_store, int num_of_bytes);
int  detectError();                  // detect errors in data packet
void reportStateError(char* location,
                      FSTK_stations station_num);
int  checkReadError(FSTK_stations station_num, char* source,
                    FSTK_datatypes data_type);

//  Define what values are requested from the station.
//  Values requested should be ORed together using the mask.
//  Return TRUE if the operation is successful.
```

```
   //  See page 97-111, the 3SPACE USER'S MANUAL.
   int setDataTypes(FSTK_stations station_num, short mask);

   // Set the state of the station and return TRUE if successful.
   //  See page 128-131, the 3SPACE USER'S MANUAL.
   //
   // Note that when the state update is from FALSE (INACTIVE) to TRUE
   // (ACTIVE), a call
   // to setDataTypes() should follow the call to set_state() to
   // specify the data types.  By default, FSTK_DEFAULT_MASK is used.
   int setState(FSTK_stations station_num, int active_flag);

   // alignment reference frame functions:
   void getAlignment(FSTK_stations station_num,
                     float origin[3],
                     float x_point[3], float y_point[3]);
   int  setAlignment(FSTK_stations station_num,
                     const float origin[3],
                     const float x_point[3],
                     const float y_point[3]);
   int  resetAlignment(FSTK_stations station_num);

   // boresight function:

   int  resetBoresight(FSTK_stations station_num);

   // active hemisphere functions.
   void getHemisphere(FSTK_stations station_num, float zenith[3]);
   int  setHemisphere(FSTK_stations station_num, const float zenith[3]);
   int  resetHemisphere(FSTK_stations station_num);

   // position measurement units. FSTK_CENTIMETER is default.
   int setUnits(FSTK_units pos_units);
   inline FSTK_units getUnits() const {
      return(units);
   }

public:
   // This constructor expects the name of a configuration file for the
   // FASTRACK and a flag list indicating data types desired.
   FastrakClass(ifstream &config_fileobj,
                short datatype_flags = FSTK_DEFAULT_MASK);
   ~FastrakClass();              // destructor

   // Return true if initializing the FSTK is successful.
   inline int exists() const {
      return valid_flag;
   }

   // In multiprocess mode, suspend and resume the execution of the
   // parallel polling process, the data producer. During the
   // suspension, no new data is produced.  Return TRUE if the
   // the is successful.
```

```
   int suspend();
   void resume();

   // Get the state of the station.
   inline int getState(FSTK_stations station_num) const {
      return(active_state[station_num]);
   }

   // move data to second buffer (reduces lock overhead)
   void copyBuffer();

   // Read the specified type of data from the specified station.
   // For a successful read, data_dest[] contains the result.
   // Note that data_dest[] must be a 4-element array for quaternions;
   // for the other types of data, data_dest[] is a 3-element array.
   // Return TRUE when the read is successful.
   int readData(FSTK_stations station_num, FSTK_datatypes data_type,
                float data_dest[]);

   // Read a homogeneous transformation matrix of the sensor with
   // respect to the transmitter. For a successful read, the upper left
   // 3x3 submatrix of matrix[][] contains the rotation matrix
   // constructed from the quaternion, euler angles, or X-, Y- and
   // Z-directional cosines (depending on which type was selected) of
   // the station which results in the X-cosine in the first row,
   // Y-cosine in the second, and Z-cosine in the third; if
   // FSTK_COORD_TYPE has been selected, the last column contains the
   // position of the station, otherwise, it is filled with 0.  Return
   // TRUE if the read is succesful. Otherwise, return FALSE.
   int getHMatrix(FSTK_stations station_num, float Hmatrix[4][4]);

   // Read the current position and orientation of the station together.
   // On a successful return, posit[] contains the position and orient[]
   // contains the orientation of the station. The type of the
   // orientation, euler-angle and quaternion, is determined by
   // orient_type.  Note that if orient_type is FSTK_EULER_TYPE, orient
   // is a 3-element array.  Otherwise, it must be a 4-element array.
   // Return TRUE if the read is succesful. Otherwise, return FALSE.
   int getPosOrient(FSTK_stations station_num,
                    FSTK_datatypes orient_type,
                    float pos[3], float orient[]);

   // Boresight functions
   void getBoresight(FSTK_stations station_num, float orient[3]);
   int  setBoresight(FSTK_stations station_num, const float orient[3]);
};

#endif
```

```
// ********************************************************************
// File    : FastrakClass.cc
// Author  : Scott McMillan
//         : Naval Postgraduate School
//         : Code CSMs
//         : Monterey, CA 93943
//         : mcmillan@cs.nps.navy.mil
// Project : NPSNET - Individual Combatants/Insertion of Humans into VEs
// Created : August 1995
// Summary : This file contains the declarations for a C++ class to
//         : manage the Polhemus 3Space Fastrak.
//         :
//         : For detailed information on the operation of the Fastrak,
//         : refer to the 3SPACE USER'S MANUAL.
//         :
//         : This program was based on the ISOTRACK program written by
//         : Paul T. Barham in Sept. 1993 for single sensor case.  Major
//         : modifications have been made to adapt to multiple sensor
//         : case.  The resulting code, written by Jiang Zhu in
//         : Jan. 1995, underwent another major modification to support
//         : binary data in continuous mode.
// ********************************************************************

/*
 * Copyright (c) 1995,
 *    Naval Postgraduate School
 *    Computer Graphics and Video Laboratory
 *    NPSNET Research Group
 *    npsnet@cs.nps.navy.mil
 *
 *
 *    Permission to use, copy, and modify this software and its
 * documentation for any non-commercial purpose is hereby granted
 * without fee, provided that (i) the above copyright notices and the
 * following permission notices appear in ALL copies of the software
 * and related documentation, and (ii) The Naval Postgraduate School
 * Computer Graphics and Video Laboratory and the NPSNET Research Group
 * be given written credit in your software's written documentation and
 * be given graphical credit on any start-up/credit screen your
 * software generates.
 *    This restriction helps justify our research efforts to the
 * sponsors who fund our research.
 *
 *    Do not redistribute this code without the express written
 * consent of the NPSNET Research Group. (E-mail communication and our
 * confirmation qualifies as written permission.) As stated above, this
 * restriction helps justify our research efforts to the sponsors who
 * fund our research.
 *
 *    This software was designed and implemented at U.S. Government
 * expense and by employees of the U.S. Government.  It is illegal
 * to charge any U.S. Government agency for its partial or full use.
 *
```

```
 * THE SOFTWARE IS PROVIDED “AS IS” AND WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS, IMPLIED OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY
 * WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
 *
 * E-Mail addresses:
 *    npsnet@cs.nps.navy.mil
 *       General code questions, concerns, comments, requests for
 *       distributions and documentation, and bug reports.
 *    npsnet-info@cs.nps.navy.mil
 *       Contact principle investigators.
 *       Overall research project information and funding.
 *       Requests for demonstations.
 *
 * Thank you to our sponsors:  ARL, STRICOM, TRAC, ARPA and DMSO.
 */

#include <stdlib.h>          // C standard library stuff
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <bstring.h>
#include <iostream.h>        // For C++ standard I/O stuff
#include <fstream.h>         // For C++ file I/O stuff
#include <unistd.h>          // For standard Unix read, write stuff
#include <errno.h>
#include <fcntl.h>           // For file constant definitions and flags
#include <termio.h>          // For terminal I/O stuff
#include <termios.h>         // For terminal I/O stuff
#include <sys/types.h>       // For system type stuff
#include <sys/prctl.h>       // For process control stuff
#include <sys/signal.h>      // For process signal stuff

#include “FastrakClass.h”

//#include “jointmods.h”

// the file for creating the shared data arena used by parallel
// processes.
#define IARENA_FILE “/tmp/fastrak.arena.data”

// the permission option to chmod command to alter the permissions on
// the arena file to be read and written by everyone.
#define ARENA_PERMISSIONS 0666

// other convenient constants

//#define DEBUG         1
#define BELL   (char) 7

#define FSTK_X        0
#define FSTK_Y        1
#define FSTK_Z        2
#define FSTK_AZ       0        // azimuth
```

```
#define FSTK_EL       1        // elevation
#define FSTK_RO       2        // roll

#define RTOD 180.0/M_PI
#define DTOR M_PI/180.0

// local functions
static void f_sig_handler(int sig, ...);

//---------------------------------------------------------------------
//   Function: report_syserr
//    Returns:
// Parameters:
//    Summary: Report system errors
//---------------------------------------------------------------------
static void report_syserr(char* err_info, char* location)
{
   cerr << BELL << "Error in " << err_info << ".\n"
        << "  Error Number in *FastrakClass.cc*" << location << ": "
        << errno << ";\n";
   perror("  Error Message");
}  // end report_syserr()


//---------------------------------------------------------------------
//   Function: pollContinuously
//    Returns:
// Parameters:
//    Summary: This is the data producer which is called in multiprocess
//           : mode in a process running in parallel with the one that
//           : calls the constructor and most of other methods of
//           : "FastrakClass". It continuously reads the Fastrak for
//           : data and stores the data in the datarec. This process runs
//           : until the "kill_flag" is set by the consumer.
//---------------------------------------------------------------------
void pollContinuously(void* tracker_object)
{
   static     void (*temp_signal)(...);
   FastrakClass *tracker = (FastrakClass*)tracker_object;

   // The following call is necessary for this child process to actually
   // respond to SIGHUP signals when the parent process dies.

   prctl(PR_TERMCHILD);

   if ((signal ( SIGTERM, (void (*)(...))f_sig_handler )) == SIG_ERR)
            perror("FastrakClass:\tError setting SIGTERM handler -\n   ");

   if ((signal ( SIGHUP, (void (*)(...))f_sig_handler )) == SIG_ERR)
            perror("FastrakClass:\tError setting SIGHUP handler -\n   ");

   if ( (temp_signal = signal ( SIGINT, (void (*)(...))f_sig_handler ))
      != SIG_DFL ) {
```

```
      if ((signal ( SIGINT, SIG_IGN )) == SIG_ERR)
         perror("FastrakClass:\tError setting SIGINT ignore -\n   ");
   }

   if ((signal ( SIGQUIT, (void (*)(...))f_sig_handler )) == SIG_ERR)
      perror("FastrakClass:\tError setting SIGQUIT handler -\n   ");

#if DEBUG
   cerr << "Fastrak polling process initiated." << endl;
#endif

   if (write(tracker->port_fd, "C", 1) != 1){
      report_syserr("sending FASTRAK C command",
                    "pollContinuously");
   }

   while (!tracker->kill_flag) {
      tracker->getPacket();
   } // end while loop

   if (write(tracker->port_fd, "c", 1) != 1){
      report_syserr("sending FASTRAK c command",
                    "pollContinuously");
   }

   tracker->kill_flag = FALSE;

#if DEBUG
   cerr << "Fastrak polling process terminated." << endl;
#endif

   exit(0);
} // end pollContinuously()


//---------------------------------------------------------------------
//   Function: f_sig_handler
//    Summary:
// Parameters:
//    Returns:
//---------------------------------------------------------------------
void f_sig_handler(int sig, ...)
{
   if ((signal ( sig, SIG_IGN )) == SIG_ERR) {
      perror("FastrakClass:\tSignal Error -\n   ");
   }

   switch ( sig ) {
      case SIGTERM:
         exit(0);
         break;
      case SIGHUP:
         if ( getppid() == 1 ) {
```

```
            cerr << "DEATH NOTICE:" << endl;
            cerr << "\tParent process terminated."
                 << endl;
            cerr << "\tFastrakClass terminating to prevent orphan process."
                 << endl;
            exit(0);
         }
         break;
      case SIGINT:
         cerr << "DEATH NOTICE: FastrakClass exitting due to interrupt "
              << "signal." << endl;
         exit(0);
         break;
      case SIGQUIT:
         cerr << "DEATH NOTICE: FastrakClass exitting due to quit "
              << "signal." << endl;
         exit(0);
         break;
      default:
         cerr << "DEATH NOTICE: FastrakClass exitting due to signal:  "
              << sig << endl;
         exit(0);
         break;
   }
   signal ( sig, (void (*)(...))f_sig_handler );
}

//========================================================================
// FastrakClass class methods
//========================================================================

//------------------------------------------------------------------------
//   Function: FastrakClass::FastrakClass
//    Returns:
// Parameters:
//    Summary: constructor for FastrakClass object
//------------------------------------------------------------------------
FastrakClass::FastrakClass(ifstream &config_fileobj, short datatype_flags)
{
   // Initialize instance variables.
   initState();

   // Read in the configurations and open the FASTRACK port.
   if ((valid_flag = readConfig(config_fileobj)) == TRUE) {
      valid_flag = openIOPort();
   }

   if (valid_flag) {
      // the ^Y command for resetting FASTRAK
      // Refer to pp 76, the 3 SPACE USER'S MANUAL for details.
      static char command[] = { 0x19 };
```

```
      cerr << "Initializing Fastrak to start-up state." << endl
           << "    This takes about 10 seconds...";
      if (write(port_fd, command, 1) == -1) {
         valid_flag = FALSE;
         report_syserr("sending FASTRAK ^Y command",
                       "FastrakClass::FastrakClass");
      }
      else {
         sleep(2);
         for (int i=9; i>-1; i--) {
            cerr << i << ' ';
            sleep(1);
         }
         cerr << endl;
      }
   }

   if ((valid_flag) && (valid_flag = checkState())) {
      valid_flag = setUnits(FSTK_CENTIMETER);
      for (int i = 0; i < FSTK_NUM_STATIONS; i++) {
         if ((valid_flag) && (active_state[i])) {
            valid_flag = setDataTypes((FSTK_stations)i, datatype_flags);
         }
      }
   }

   for (int station_num = 0; station_num < FSTK_NUM_STATIONS; station_num++) {
      if ((valid_flag) && (active_state[station_num])) {
         valid_flag = setHemisphere((FSTK_stations)station_num,
                                    hemisphere[station_num]);
         if (valid_flag) {
            valid_flag = setAlignment((FSTK_stations)station_num,
                                      alignment[station_num][0],
                                      alignment[station_num][1],
                                      alignment[station_num][2]);
         }
      }
   }

   // mcmillan - 950814 - new code to read IEEE binary format data
   if (valid_flag) {
      // Enable binary output format from the FASTRAK.
      // Refer to pp 114, the 3 SPACE USER'S MANUAL.
      if (write(port_fd, "f", 1) != 1) {
         report_syserr("sending FASTRAK f command",
                       "FastrakClass::FastrakClass");
         valid_flag = FALSE;
      }
   }
```

```
   if (valid_flag) {
      // Initialize the multiprocess mode, i.e., creating the locks
      // and semaphors and sprocing the Fastrak data producer process.
      valid_flag = initMultiprocessing();
   }

   if (valid_flag) {
      is_polling_flag = TRUE;
   }

#if DEBUG
   cerr << "The FASTRAK object constructed.\n";
   if (valid_flag) {
      cerr << "The FASTRAK intialization is successful." << endl;
   }
   else {
      cerr << "The FASTRAK intialization is unsuccessful." << endl;
   }
#endif

}  // end FastrakClass::FastrakClass()


//---------------------------------------------------------------------
//   Function: FastrakClass::~FastrakClass
//    Returns:
// Parameters:
//    Summary: destructor for FastrakClass class
//---------------------------------------------------------------------
FastrakClass::~FastrakClass()
{
#if DEBUG
      cerr << "Fastrak destructor called.\n";
#endif

   if (valid_flag) {
      // In multiprocess mode, signal the producer process to die.
      // Then, free the lock and semaphore.

      if (parpoll_pid != -1) {
         kill_flag = TRUE;
         if (!is_polling_flag) usvsema(paramsema);
         while (kill_flag);
         sleep(1);

         parpoll_pid = -1;
         usfreelock(datalock, arena);
         usfreesema(paramsema, arena);
      }
```

```
      // Flush all characters from the serial port and then close it.
      tcflush (port_fd, TCIOFLUSH);
      close (port_fd);
      valid_flag = FALSE;
   }
} // FastrakClass::~FastrakClass()


//-----------------------------------------------------------------------
//   Function: initState
//    Returns:
// Parameters:
//    Summary: Initialize instance variables to their default states.
//-----------------------------------------------------------------------
void FastrakClass::initState()
{
   bzero(read_buffer, BUFFER_SIZE);
   read_index = 0;
   max_datarec_size = 0;

   parpoll_pid = -1;
   param_set_flag = FALSE;
   is_polling_flag = FALSE;
   data_ready_flag = FALSE;
   kill_flag = FALSE;

   // Initialize hemispheres and alignments.
   // Refer to pp 42 - 50 and pp 88 - 92, the 3 SPACE USER'S MANUAL
   // for the default values.
   for (int station_num = 0; station_num < FSTK_NUM_STATIONS; station_num++) {
      hemisphere[station_num][FSTK_X] = 1.0;
      hemisphere[station_num][FSTK_Y] = 0.0;
      hemisphere[station_num][FSTK_Z] = 0.0;

      boresight[station_num][FSTK_AZ] = 0.0;
      boresight[station_num][FSTK_EL] = 0.0;
      boresight[station_num][FSTK_RO] = 0.0;

      // origins
      alignment[station_num][0][FSTK_X] = 0.0;
      alignment[station_num][0][FSTK_Y] = 0.0;
      alignment[station_num][0][FSTK_Z] = 0.0;

      // X directions
      alignment[station_num][1][FSTK_X] = 1.0;
      alignment[station_num][1][FSTK_Y] = 0.0;
      alignment[station_num][1][FSTK_Z] = 0.0;

      // Y directions
      alignment[station_num][2][FSTK_X] = 0.0;
      alignment[station_num][2][FSTK_Y] = 1.0;
      alignment[station_num][2][FSTK_Z] = 0.0;
```

```
      // Initialize data record parameters.
      active_state[station_num] = FALSE;
      datarec_size[station_num] = 0;
      bzero(datarec_buf[station_num], MAX_PACKET_SIZE);
      bzero(datarec[station_num], MAX_PACKET_SIZE);

      for (int type_num = 0; type_num < FSTK_NUM_DATATYPES; type_num++)
         datatype_start[station_num][type_num] = -1;
   }

   fstk_packet_size = 0;
}  // end initState()


//-----------------------------------------------------------------------
//   Function: readConfig
//    Returns: TRUE if the read is successful. Otherwise, return FALSE.
// Parameters:
//    Summary: Read the configuration file for the FASTRAK.  Called by a
//           : "FastrakClass" constructor to do initialization. They should
//           : not be called elsewhere.
//-----------------------------------------------------------------------

#define MAX_CONFIGFILE_LINESIZE        255
#define CONFIGFILE_COMMENT_CHAR        '#'

int FastrakClass::readConfig(ifstream &config_fileobj)
{
   int success = TRUE;
   char tmp_str[MAX_CONFIGFILE_LINESIZE];
   int station_num;

   while (config_fileobj >> tmp_str)
   {
      // When a comment char is read, skip the rest of the line.
      if (tmp_str[0] == CONFIGFILE_COMMENT_CHAR) {
         config_fileobj.getline(tmp_str, MAX_CONFIGFILE_LINESIZE);
      }
      else if (strncmp(tmp_str, "PORT", 4) == 0) {
         config_fileobj >> port_name;
      }
      else if (strncmp(tmp_str, "WANTED_STATIONS", 8) == 0) {
         int state;
         for (station_num = 0; station_num < FSTK_NUM_STATIONS;
                               station_num++) {
            config_fileobj >> state;
            active_state[station_num] = state;
         }
      }
```

```
      else {
         char param_str[30];
         int i, j;
         for (int station_num = 0; station_num < FSTK_NUM_STATIONS;
                                   station_num++) {
            sprintf(param_str, "STATION%d_PARAM", station_num+1);

            do {
                 if (tmp_str[0] == CONFIGFILE_COMMENT_CHAR)
                   config_fileobj.getline(tmp_str, MAX_CONFIGFILE_LINESIZE);
               else if (strncmp(tmp_str, param_str, 10) == 0) {
                  char param_name[30];

                  config_fileobj >> param_name;
                  for (i = 0; i < 3; i++)
                     config_fileobj >> hemisphere[station_num][i];

                  for (i = 0; i < 3; i++) {
                     config_fileobj >> param_name;
                     for (j = 0; j < 3; j++)
                        config_fileobj >> alignment[station_num][i][j];
                  }

                  config_fileobj >> tmp_str;
                  break;
               }
               else {
                  success = FALSE;
                     cerr << BELL << "Error in reading config file.\n"
                       << "  in *FastrakClass.cc*FastrakClass::readConfig;"
                       << " illegal string: " << tmp_str << endl;
               }
            } while (config_fileobj >> tmp_str);

         } // end for
      } // end if
   } // end while

#if DEBUG
   int i, j;
   cerr << "** readConfig:\n" << "   FASTRAK port: " << port_name << "\n"
        << "   Stations requested: ";

   for (i = 0; i < FSTK_NUM_STATIONS; i++)
      cerr << active_state[i] << " ";
   cerr << endl << endl;

   for (i = 0; i < FSTK_NUM_STATIONS; i++) {
      cerr << "STATION" << i+1 << "_PARAM:\n";
      cerr << "  hemisphere:\t";
      for (j = 0; j < 3; j++)
         cerr << hemisphere[i][j] << " ";
      cerr << "\n  origin:\t";
```

```
      for (j = 0; j < 3; j++)
         cerr << alignment[i][0][j] << " ";
      cerr << "\n  x_point:\t";
      for (j = 0; j < 3; j++)
         cerr << alignment[i][1][j] << " ";
      cerr << "\n  y_point:\t";
      for (j = 0; j < 3; j++)
         cerr << alignment[i][2][j] << " ";
      cerr << endl;
   }
   cerr << endl;

   cerr << "FASTRAK configuration parameters read.\n";
#endif

   return (success);
}  // end FastrakClass::readConfig()


//-------------------------------------------------------------------------
//   Function: openIOPort
//    Returns: TRUE for a successful opening. Otherwise, return FALSE.
// Parameters:
//    Summary: Open the FASTRAK io-port
//-------------------------------------------------------------------------
int FastrakClass::openIOPort()
{
   int success = TRUE;

   // Test to see if the FASTRAK if on.
   if ((port_fd = open(port_name, O_RDWR|O_NONBLOCK)) == -1) {
      success = FALSE;
      report_syserr("opening the Fastrak port",
                    "FastrakClass::openIOPort");
   }
   else {
      char command[5], buffer[100];
      strcpy(command, "l1\r");

      if (write(port_fd, command, strlen(command)) == -1) {
         success = FALSE;
         report_syserr("sending FASTRAK l command",
                       "FastrakClass::openIOPort");
      }

      sleep(1);

      if ((success) && (read(port_fd, buffer, 100) == -1)) {
         success = FALSE;
         report_syserr("reading FASTRAK",
                       "FastrakClass::openIOPort");
      }
```

```
      close(port_fd);
   }

   // Do a blocking read when polling the FASTRAK for data.
   if ((success) && ((port_fd = open(port_name, O_RDWR)) == -1)) {
      success = FALSE;
      report_syserr("opening the Fastrak port",
                    "FastrakClass::openIOPort");
   }
   else if (success)  {
      struct termio term;
      memset(&term, 0, sizeof(term));

      term.c_iflag = IXOFF;            /* FIXME */
      term.c_oflag = 0;
      term.c_cflag = B9600|CS8|CLOCAL|CREAD|HUPCL;
      term.c_lflag = 0;
      term.c_line  = 0;     // LDISC1;
      term.c_cc[VMIN] = 0;
      term.c_cc[VTIME] = 5;

      if (ioctl(port_fd, TCSBRK, 0) == -1) {
         success = FALSE;
         close (port_fd);
         report_syserr("sending a BREAK to the Fastrak port",
                       "FastrakClass::openIOPort");
      }
      else if (ioctl(port_fd, TCSETAF, &term) == -1) {
         success = FALSE;
         close (port_fd);
         report_syserr("setting the Fastrak port parameters",
                       "FastrakClass::openIOPort");
      }
   }

   // Just in case the fastrak was accidentally left in continuous mode?
   if (success) {
      char data[100];
      int nbr;
      while ((nbr = read(port_fd, data, 100)) > 0) {
         //cerr << "Warning: cleared " << nbr << "bytes from Fastrak buffer"
         //     << " in openIOport." << endl;
         write(port_fd, "c", 1);
         //debugData(data,nbr);
      }
   }
```

```
   if (success) {
      if (tcflush (port_fd, TCIOFLUSH) == -1) {
         success = FALSE;
         close (port_fd);
         report_syserr("flushing the Fastrak port",
                       "FastrakClass::openIOPort");
      }
   }

#if DEBUG
   cerr << "FASTRAK io-port, " << port_name << ", opened." << endl;
#endif

   return(success);
} // end FastrakClass::openIOPort()


//-------------------------------------------------------------------------
//   Function: checkState
//    Returns: TRUE if all the requested stations are available.
//           : Otherwise, return FALSE.
// Parameters:
//    Summary: Check for the availability of the FASTRAK stations.
//-------------------------------------------------------------------------
int FastrakClass::checkState()
{
   int success = TRUE;
   char command[5];
   char data[100];

   // Construct the "l" command to get the states of the stations.
   // Refer to pp 128 - 131, the 3 SPACE USER's MANUAL for details.
   // Choose any station to get the states for all stations.
   strcpy(command, "l1\r");

   if (write(port_fd, command, strlen(command)) == -1) {
      success = FALSE;
      report_syserr("sending FASTRAK l command",
                    "FastrakClass::checkState");
   }

   // Find out which station is active by hardware configuration.
   if (success == TRUE) {

      // 951002 - mcmillan - IMPORTANT BUG FIX:
      // do raw tty processing to get the answer because on the faster
      // machines and especially the onyx platforms the read occurs
      // sooner than the data is ready.
      const int NUM_BYTES = 9;
      const int MAX_POLL_RETRIES = 100000;
      int count = 0;
      int num_tries = 0;
      int nbr;
```

```
      while ((count < NUM_BYTES) &&
             (num_tries < MAX_POLL_RETRIES) &&
             ((nbr = read(port_fd, &data[count], 1)) != -1)) {
         num_tries++;
         count += nbr;
      }

      if (count != NUM_BYTES) {
         success = FALSE;
         if (num_tries == MAX_POLL_RETRIES) {
            cerr << BELL << "Error: too many retries reading fstk port\n";
         }
         else {
            cerr << BELL << "Error: fstk port read failed\n";
         }
         cerr << "  in *FastrakClass.cc*FastrakClass::checkState\n";
      }

#if DEBUG
      debugData(data, 9);
#endif
   }

   if (success == TRUE) {
      for (int i = 0; i < FSTK_NUM_STATIONS; i++) {
         active_setting[i] = (data[i + FSTK_HEADER_SIZE] == '1');

         if (active_setting[i]) {
            if (!active_state[i])
               setState((FSTK_stations)i, FALSE);
         }
         else {
            if (active_state[i]) {
               success = FALSE;
               cerr << BELL
                    << "Error in setting FASTRAK station state.\n"
                    << "  in *FastrakClass.cc*FastrakClass::checkState\n"
                    << "  Station" << i+1 << " is required to be active.\n"
                    << "  However, it is not set to be active by the"
                    << "  hardware switch.\n";
               debugData(data, 9);
            }
         }
      }  // end for
   }
   else {
      report_syserr("reading FASTRAK", "FastrakClass::checkState");
   }

#if DEBUG
   cerr << "The states of the FASTRAK stations checked." << endl;
#endif
```

```
   return(success);
}  // end FastrakClass::checkState()


//---------------------------------------------------------------------
//   Function: initMultiprocessing
//    Returns: TRUE if the initialization is successful. Otherwise,
//           : return FALSE.
// Parameters:
//    Summary: Initialize for multiprocess mode, including getting locks
//           : and semaphores and sprocing a child process for polling
//           : the FASTRAK.
//---------------------------------------------------------------------
int FastrakClass::initMultiprocessing()
{
   int success = TRUE;

   // Create an arena file to get the needed lock and semaphore
   if ((arena = usinit(IARENA_FILE)) == NULL) {
      success = FALSE;
      report_syserr("getting an arena file",
                    "FastrakClass::initMultiprocessing");
   }
   else {
      // Set up the arena file with read and write permissions for
      // everyone.
      if (usconfig(CONF_CHMOD, arena, ARENA_PERMISSIONS) == -1) {
         success = FALSE;
         report_syserr("configuring an arena",
                       "FastrakClass::initMultiprocessing");
      }

      // Create a lock to provide mutual exclusive access to the data
      // buffers.  Refer to getPacket() for more info.
      if (success && ((datalock = usnewlock(arena)) == NULL)) {
         success = FALSE;
         report_syserr("creating a lock",
                       "FastrakClass::initMultiprocessing");
      }
      else {
         usinitlock(datalock);
      }

      // Create a binary semaphore for providing mutual exclusions so
      // that when the data record parameters are being set, the data
      // producer waits until the setting finishes.  Refer to
      // getPacket() and setDataTypes() for more info.
      if (success &&
          ((paramsema = usnewsema(arena, 1)) == NULL))  {
         success = FALSE;
         report_syserr("creating a binary semaphore",
                       "FastrakClass::initMultiprocessing");
      }
```

```
      // Fork the parallel data producer as a lightweight thread which
      // shares the data space with its parent process that is the
      // consumer of the FASTRAK data.
      if (success)  {
         parpoll_pid = sproc(pollContinuously, PR_SALL, (void *)this);
         if (parpoll_pid == -1) {
            success = FALSE;
            report_syserr("spawning the producer process",
                          "FastrakClass::initMultiprocessing");
         }
         else {
            signal(SIGCLD,SIG_IGN);
#if DEBUG
            cerr << "Fastrak poll process spawned: pid = "
                 << parpoll_pid << endl;
#endif
         }
      }
   }  // if arena

   return (success);
} // end FastrakClass::initMultiprocessing()


//---------------------------------------------------------------------
//   Function: getPacket
//    Returns:
// Parameters:
//    Summary: Read a packet from the FASTRAK.  In single process mode,
//           : it is called when the FASTTRACk user requests a data. In
//           : multiprocess mode, it is continuously called by the data
//           : producer, pollContinuously(), in a light weight process.
//---------------------------------------------------------------------
void FastrakClass::getPacket()
{
   int full_buffer;

   // The following piece of code is a critical section in multiprocess
   // mode.  During a read operation of the FASTRAK, the data record
   // parameters cannot be changed, e.g, through setDataTypes()
   // which runs in parallel with this method.
   if (parpoll_pid != -1) {
      if (uspsema(paramsema) == -1) { // entering the critical section
         report_syserr("getting semaphore",
                       "FastrakClass::getPacket");
      }
   }
```

```
   else {
      // In serial mode, ask for a packet from the FASTRAK.
      // Refer to pp 120, the 3 SPACE USER'S MANUAL.
      if (write (port_fd, "P", 1) != 1) {
         report_syserr("sending FASTRAK P command",
                       "FastrakClass::getPacket");
      }
   }

   // ================================================
   // keep reading until device buffer is exhuasted
   do {
      unsigned int num_bytes_to_read = BUFFER_SIZE - read_index;
      int num_bytes_read = read(port_fd,
                                &read_buffer[read_index],
                                num_bytes_to_read);

      if (num_bytes_read > num_bytes_to_read) {
         cerr << "Error: fstk read too many bytes (nbr, nbtr): ("
              << num_bytes_read << "," << num_bytes_to_read
              << ").\n";
      }

      if (num_bytes_read == num_bytes_to_read) {
         full_buffer = TRUE;
         cerr << "Warning: fstk read max bytes: "
              << num_bytes_read << ".\n";
      }

      // process the data read
      if (num_bytes_read > 0) {
         unsigned int index = 0;
         read_index += num_bytes_read;

         // while there is enough information for a packet from a
         // single station process the data:
         while (!kill_flag && ((read_index - index) > max_datarec_size)) {

            // make sure header info is the first few bytes
            if ((read_buffer[index] == 0x30) &&
                ((read_buffer[index+1]&0xf0) == 0x30) &&
                (((int) (read_buffer[index+1]&0x0f) - 1) < 4)) {

               int station_num = (int) (read_buffer[index+1]&0x0f) - 1;

               // ***************************
               // entering the critical section
               if (ussetlock(datalock) == -1)
                  report_syserr("getting lock", "packetizer");

               memcpy(datarec_buf[station_num], &read_buffer[index],
                      datarec_size[station_num]);
               data_ready_flag = TRUE;
```

```
               usunsetlock(datalock);        // unlocking
               // exiting the critical section
               // ***************************

               index += datarec_size[station_num];
            }
            else { // find the header info and hopefully resynch.
               cerr << "Warning: resynching fstk "
                    << "(index, read_index): = ("
                    << index << ", " << read_index << ")\n";
               while ((index < (read_index-1)) &&
                      !((read_buffer[index] == 0x30) &&
                        ((read_buffer[index+1]&0xf0) == 0x30) &&
                        (((int) (read_buffer[index+1]&0x0f) - 1) < 4))) {
                  cerr << hex << (int) read_buffer[index] << dec
                       << '|';
                  index++;
               }
               cerr << hex << (int) read_buffer[index] << " "
                    << (int) read_buffer[index+1] << dec
                    << '|' << " index: " << index << endl;
            }
         } // while read_index-index

         // when done, shift the rest of the buffer down to the
         // beginning.
         if (index != read_index) {
            if (index > read_index) {
               cerr << "Error: fstk shifting too many bytes ("
                    << read_index << "-" << index << ").\n";
            }
            else {
               memcpy(read_buffer, &read_buffer[index],
                      read_index-index);
            }
         }
         read_index -= index;
         index = 0;
      } // if num_bytes_read

   } while (full_buffer && !kill_flag);

   // exiting the critical section
   if (parpoll_pid != -1) usvsema(paramsema);
} // end FastrakClass::getPacket()
```

```
//---------------------------------------------------------------------
//   Function: sendCommand
//    Summary: Send a command to the FASTRACK.
// Parameters: command string, its length, and initiating fcn name
//    Returns: TRUE if the operation is successful. Otherwise, FALSE.
//---------------------------------------------------------------------
int FastrakClass::sendCommand(char* command, int length, char* source)
{
   int success = TRUE;

   (source);
#if DEBUG
   cerr << “command from “ << source << “:”
        << command << endl;
#endif

   // This is a critical section in multiprocess mode.
   if (parpoll_pid != -1) {
      if (uspsema(paramsema) == -1) {  // entering the critical section
         success = FALSE;
         report_syserr(“getting semaphore”, “FastrakClass::sendCommand”);
      }
   }

   if (write(port_fd, command, length) == -1) {
      success = FALSE;
      report_syserr(“sending FASTRACK command”, “FastrakClass::sendCommand”);
   }

   data_ready_flag = FALSE;  // See getPacket() for when it is set to TRUE.

   // exiting the critical section
   if (parpoll_pid != -1) usvsema(paramsema);

#if DEBUG
   cerr << “FASTRAK command sent: source being “ << source << “.\n”;
#endif

   return(success);
}  // FastrakClass::sendCommand()


//---------------------------------------------------------------------
//   Function: convertData
//    Returns:
// Parameters: data record ptr, number of elements, output vector
//    Summary: Convert a DOS ordered bytes to Unix order (reverse)
//---------------------------------------------------------------------
void FastrakClass::convertData(char* data, int num_floats,
                               float data_dest[])
{
   char *ptr = data;
   char *fptr = (char *) data_dest;
```

```
   for (int i=0; i<num_floats; i++) {
      *(fptr+3) = *ptr++;
      *(fptr+2) = *ptr++;
      *(fptr+1) = *ptr++;
      *(fptr)   = *ptr++;

      fptr += 4;
   }
}  // FastrakClass::convertData()


//---------------------------------------------------------------------
//   Function: convert16BITData
//    Returns:
// Parameters: data record ptr, number of elements, scale, output vector
//    Summary: Convert a Polhemus's 16BIT format to IEEE floating point
//---------------------------------------------------------------------
void FastrakClass::convert16BITData(char* data, int num_floats,
                                    float scale, float data_dest[])
{
   char *ptr = data;
   char lobyte;
   char hibyte;
   int sign_flag, num;

   for (int i=0; i<num_floats; i++) {
      lobyte = *ptr++;
      hibyte = *ptr++;
      sign_flag = (int) hibyte&0x040;
      num = ((hibyte << 7) + (lobyte&0x7f))&0x001fff;
      if (sign_flag) {
         num -= 0x02000; // 14 bit 2's complement conversion.
      }

      data_dest[i] = scale*num;
   }
}  // FastrakClass::convert16BITData()
```

```
//---------------------------------------------------------------------
//   Function: debugData
//    Returns:
// Parameters:
//    Summary: Write num_of_bytes starting from data_store as characters
//           : to cerr. This is a convienience function used to examine
//           : the data packet read in from FASTRAK.
//---------------------------------------------------------------------
void FastrakClass::debugData(char *data_store, int num_of_bytes)
{
   cerr << "Record length: " << num_of_bytes << "\n";
   cerr << "|";
   for (int i = 0; i < num_of_bytes; i++)
      cerr << hex << (int) data_store[i] << dec << "|";
   cerr << "\n";
}


//---------------------------------------------------------------------
//   Function: detectError
//    Returns:
// Parameters:
//    Summary:
//---------------------------------------------------------------------
int FastrakClass::detectError()
{
   char *station_data;
   int success = TRUE;

   // This method should only be used in critical sections. As a result,
   // no semaphore protection is needed here.
   for (int i = 0; i < FSTK_NUM_STATIONS; i++) {
      station_data = datarec[i];

      switch (*station_data) {
         case '0': break; // No error for data record - do nothing
         case '2': cerr << "Fastrak Type 2 Record received.\n";
            break;
         case 'A': cerr << BELL
                        << "HARDWARE ERROR found in Fastrak station"
                        << i+1
                        << "  EPROM CHECK SUM. (character A)" << endl;
                        break;
         case 'C': cerr << BELL
                        << "HARDWARE ERROR found in Fastrak station"
                        << i+1
                        << "  RAM TEST. (character C)" << endl;
                        break;
         case 'S': cerr << BELL
                        << "HARDWARE ERROR found in Fastrak station"
                        << i+1
                        << "  SELF-CALIBRATION. (character S)" << endl;
                        break;
```

```
        case 'U': cerr << BELL
                       << "HARDWARE ERROR found in Fastrak station"
                       << i+1
                       << "  SOURCE/SENSOR ID PROM. (character U)"
                       << endl;
                       break;
        case 'a': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  CALCULATE TRACE OF S4TS4. (character a)"
                       << endl;
                       break;
        case 'b': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  SELF-CAL DIVIDE. (character b)" << endl;
                       break;
        case 'c': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  SELF-CAL A/D INPUT. (character c)"
                       << endl;
                       break;
        case 'd': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  SENSOR A/D INPUT. (character d)" << endl;
                       break;
        case 'e': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  OUT OF ENVELOPE. (character e)" << endl;
                       break;
        case 'f': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  SELF-CAL OFFSET OVERFLOW. (character f)"
                       << endl;
                       break;
        case 'g': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  TRMDI CALCULATION. (character g)" << endl;
                       break;
        case 'h': cerr << BELL
                       << "SOFTWARE ERROR found in Fastrak station"
                       << i+1
                       << "  PATH1. (character h)" << endl;
                       break;
```

```
         case ‘i’: cerr << BELL
                        << “SOFTWARE ERROR found in Fastrak station”
                        << i+1
                        << “  PATH2. (character i)” << endl;
                        break;
         case ‘j’: cerr << BELL
                        << “SOFTWARE ERROR found in Fastrak station”
                        << i+1
                        << “  PATH3. (character j)” << endl;
                        break;
         case ‘k’: cerr << BELL
                        << “SOFTWARE ERROR found in Fastrak station”
                        << i+1
                        << “  PATH4. (character k)” << endl;
                        break;
         case ‘l’: cerr << BELL
                        << “SOFTWARE ERROR found in Fastrak station”
                        << i+1
                        << “  SYSTEM RUNNING TOO SLOW. (character l)”
                        << endl;
                        break;
         case ‘n’: cerr << BELL
                        << “SOFTWARE ERROR found in Fastrak station” << i+1
                        << “  ATTITUDE MATRIX CALCULATION. (character n)”
                        << endl;
                        break;
         case ‘p’: cerr << BELL
                        << “SOFTWARE ERROR found in Fastrak station”
                        << i+1
                        << “  NORM OF XORVEC TOO LOW. (character p)”
                        << endl;
                        break;
         default : cerr << BELL
                        << “UNKNOWN ERROR found in Fastrak station”
                        << i+1 << “: (character “ << *station_data
                        << “) ASCII code: “
                        << int(*station_data) << endl;
                        break;
      } // end switch
   } // end for

   return(success);

}  // end detectError()
```

```
//---------------------------------------------------------------------
//   Function: reportStateError
//    Returns:
// Parameters:
//    Summary: Report errors caused by trying to use inactive stations.
//---------------------------------------------------------------------
void FastrakClass::reportStateError(char* location,
                                    FSTK_stations station_num)
{
   cerr << BELL << "Error in using station" << station_num+1 << "\n"
        << "  in *FastrakClass.cc*FastrakClass::"
        << location << "; inactive station" << endl;
}

//---------------------------------------------------------------------
//   Function: checkReadError
//    Returns: TRUE if no read-data error, else FALSE
// Parameters: station number
//    Summary: Check for read-data error.
//---------------------------------------------------------------------
int FastrakClass::checkReadError(FSTK_stations station_num,
                                 char* source, FSTK_datatypes data_type)
{
   int success = TRUE;

   if (!active_state[station_num]) {
      reportStateError(source, station_num);
      success = FALSE;
   }

   if (datatype_start[station_num][data_type] < 0) {
      cerr << BELL << "Error in reading FASTRAK station"
           << station_num+1 << ".\n"
           << "  in *FastrakClass.cc*FastrakClass::" << source
             << "; unrequested data type\n" << endl;
      success = FALSE;
   }

   if (!is_polling_flag) {
      cerr << BELL << "Error in reading FASTRAK station"
           << station_num+1 << ".\n"
           << "  in *FastrakClass.cc*FastrakClass::" << source
           << "; polling process suspended\n" << endl;
      success = FALSE;
   }

   return(success);
}  // FastrakClass::checkReadError()
```

```
//------------------------------------------------------------------------
//   Function: suspend
//    Returns: TRUE if the suspension is successful. Otherwise,  FALSE.
// Parameters:
//    Summary: In multiprocess mode, suspend the execution of the
//           : parallel polling process.
//------------------------------------------------------------------------
int FastrakClass::suspend()
{
   int success = TRUE;

   if (is_polling_flag) {
      if (parpoll_pid != NULL) {
         if (uspsema(paramsema) == -1) {
            success = FALSE;
            report_syserr("suspending the polling process",
                          "FastrakClass::suspend");
         }
         else {
            is_polling_flag = FALSE;
         }
      }
#if DEBUG
      cerr << "Parallel polling process suspended: " << success << endl;
#endif
   }
   return(success);
}


//------------------------------------------------------------------------
//   Function: resume
//    Returns:
// Parameters:
//    Summary: Resume the execution of the parallel polling process.
//------------------------------------------------------------------------
void FastrakClass::resume()
{
   if (!is_polling_flag) {
      if (parpoll_pid != NULL) usvsema (paramsema);

      is_polling_flag = TRUE;
#if DEBUG
      cerr << "Parallel polling process resumed.\n";
#endif
   }
}
```

```
//-----------------------------------------------------------------------
//   Function: setAlignment
//    Returns: TRUE if the operation is successful else, FALSE.
// Parameters: station number
//    Summary: set the alignment of a station.
//-----------------------------------------------------------------------
int FastrakClass::setAlignment(FSTK_stations station_num,
                               const float origin[3],
                               const float x_point[3],
                               const float y_point[3])
{
   if (!active_state[station_num]) {
      reportStateError("setAlignment", station_num);
      return(FALSE);
   }

   // Construct the "A" command to set the alignment of the station.
   // Refer to pp 42-49, the 3 SPACE USER's MANUAL for details.
   static char command[100];
   sprintf(command, "A%d,%.1f,%.1f,%.1f,%.1f,%.1f,%.1f,%.1f,%.1f\r",
           station_num+1,
           origin[FSTK_X], origin[FSTK_Y], origin[FSTK_Z],
           x_point[FSTK_X], x_point[FSTK_Y], x_point[FSTK_Z],
           y_point[FSTK_X], y_point[FSTK_Y], y_point[FSTK_Z]);

   int success =  sendCommand(command, strlen(command),
                              "FastrakClass::setAlignment");

   if (success == TRUE) {
      for (int i = 0; i < 3; i++)
      {
         alignment[station_num][0][i] = origin[i];
         alignment[station_num][1][i] = x_point[i];
         alignment[station_num][2][i] = y_point[i];
      }
   }

#if DEBUG
   cerr << "FASTRAK alignment set.\n";
#endif

   return(success);
}  // FastrakClass::setAlignment()
```

```
//-----------------------------------------------------------------------
//   Function: resetAlignment
//    Returns: TRUE if the operation is successful else, FALSE.
// Parameters: station number
//    Summary: Reset the alignment of a station.
//-----------------------------------------------------------------------
int FastrakClass::resetAlignment(FSTK_stations station_num)
{
   if (!active_state[station_num]) {
      reportStateError("resetAlignment", station_num);
      return(FALSE);
   }

   // Construct the "R" command to reset the alignment of the station to
   // default.  Refer to pp 50, the 3 SPACE USER's MANUAL for details.
   char command[10];
   sprintf(command, "R%d\r", station_num+1);

   int success = sendCommand(command, strlen(command),
                             "FastrakClass::resetAlignment");

   if (success == TRUE) {
      for (int i = 0; i < 3; i++)
         for (int j = 0; j < 3; j++)
            alignment[station_num][i][j] = 0.0;

      alignment[station_num][1][0] = 1.0;        // X direction
      alignment[station_num][2][1] = 1.0;        // Y direction
   }

#if DEBUG
   cerr << "FASTRAK alignment reset.\n";
#endif

   return(success);
}  // FastrakClass::resetAlignment
```

```
//-----------------------------------------------------------------------
//   Function: getAlignment
//    Returns: origin, x-axis, and y-axis vectors
// Parameters: station number
//    Summary: Get the alignment of the station.
//-----------------------------------------------------------------------
void FastrakClass::getAlignment(FSTK_stations station_num,
                                float origin[3],
                                float x_point[3], float y_point[3])
{
   for (int i = 0; i < 3; i++) {
      origin[i]  = alignment[station_num][0][i];
      x_point[i] = alignment[station_num][1][i];
      y_point[i] = alignment[station_num][2][i];
   }
}  // FastrakClass::getAlignment()


//-----------------------------------------------------------------------
//   Function: setBoresight
//    Returns: TRUE if the operation is successful else FALSE.
// Parameters: station number and orientation angles
//    Summary: Set the boresight of a station.
//-----------------------------------------------------------------------
int FastrakClass::setBoresight(FSTK_stations station_num,
                               const float orient[3])
{
   if (!active_state[station_num]) {
      reportStateError("setBoresight", station_num);
      return(FALSE);
   }

   // Construct the "G" command to establish the boresight reference
   // angles.  Refer to pp 51 - 54, the 3 SPACE USER's MANUAL for
   // details.
   char command[50];
   sprintf(command, "G%d,%.1f,%.1f,%.1f\r",
           station_num+1, orient[FSTK_AZ], orient[FSTK_EL], orient[FSTK_RO]);

   int success = sendCommand(command, strlen(command),
                             "FastrakClass::setBoresight");

   if (success == TRUE) {
      boresight[station_num][FSTK_AZ] = orient[FSTK_AZ];
      boresight[station_num][FSTK_EL] = orient[FSTK_EL];
      boresight[station_num][FSTK_RO] = orient[FSTK_RO];

      // Construct the "B" command to set the line_of_sight of the station.
      // Refer to pp 53, the 3 SPACE USER's MANUAL for details.
      sprintf(command, "B%d\r", station_num+1);
```

```
      success = sendCommand(command, strlen(command),
                            "FastrakClass::setBoresight");
   }

#if DEBUG
   cerr << "FASTRAK boresight set.\n";
#endif

   return(success);
}  // FastrakClass::setBoresight()



//-----------------------------------------------------------------------
//   Function: resetBoresight
//    Returns: TRUE if operation is successful
// Parameters: station number
//    Summary: Reset the boresight.
//-----------------------------------------------------------------------
int FastrakClass::resetBoresight(FSTK_stations station_num)
{
   if (!active_state[station_num]) {
      reportStateError("resetBoresight", station_num);

      return(FALSE);
   }

   // Construct the "b" command to reset the boresight of the station to
   // default.
   // Refer to pp 55, the 3 SPACE USER's MANUAL for details.
   char command[10];
   sprintf(command, "b%d\r", station_num+1);

   int success =
      sendCommand(command, strlen(command), "FastrakClass::resetBoresight");

   if (success == TRUE) {
      boresight[station_num][FSTK_AZ] = 0.0;
      boresight[station_num][FSTK_EL] = 0.0;
      boresight[station_num][FSTK_RO] = 0.0;
   }

#if DEBUG
   cerr << "FASTRAK boresight reset.\n";
#endif

   return(success);
}  // FastrakClass::resetBoresight()
```

```
//---------------------------------------------------------------------
//   Function: getBoresight
//    Returns: euler angles defining the boresight
// Parameters: station number
//    Summary:
//---------------------------------------------------------------------
void FastrakClass::getBoresight(FSTK_stations station_num, float orient[3])
{
   orient[FSTK_AZ] = boresight[station_num][FSTK_AZ];
   orient[FSTK_EL] = boresight[station_num][FSTK_EL];
   orient[FSTK_RO] = boresight[station_num][FSTK_RO];
}  // FastrakClass::getBoresight()


//---------------------------------------------------------------------
//   Function: setHemisphere
//    Returns: TRUE if the operation is successful. Otherwise, FALSE.
// Parameters: station number and new zenith vector
//    Summary: Set the hemisphere of a station.
//---------------------------------------------------------------------
int FastrakClass::setHemisphere(FSTK_stations station_num,
                                const float zenith[3])
{
   if (!active_state[station_num]) {
      reportStateError("setHemisphere", station_num);
      return(FALSE);
   }

   // Construct the "H" command to set the hemisphere of the station.
   // Refer to pp 88 - 92, the 3 SPACE USER's MANUAL for details.
   char command[50];
   sprintf(command, "H%d,%.1f,%.1f,%.1f\r",
           station_num+1, zenith[FSTK_X], zenith[FSTK_Y], zenith[FSTK_Z]);

   int success =
      sendCommand(command, strlen(command), "FastrakClass::setHemisphere");

   if (success == TRUE) {
      hemisphere[station_num][FSTK_X] = zenith[FSTK_X];
      hemisphere[station_num][FSTK_Y] = zenith[FSTK_Y];
      hemisphere[station_num][FSTK_Z] = zenith[FSTK_Z];
   }

#if DEBUG
   cerr << "FASTRAK hemisphere set:"
        << zenith[FSTK_X] << ", " << zenith[FSTK_Y] << ", "
        << zenith[FSTK_Z] << ".\n";
#endif

   return (success);
}  // FastrakClass::setHemisphere()
```

```
//------------------------------------------------------------------------
//   Function: resetHemisphere
//    Returns: FALSE if station inactive or reset fails, TRUE otherwise
// Parameters: station number
//    Summary: Reset the hemisphere.
//------------------------------------------------------------------------
int FastrakClass::resetHemisphere(FSTK_stations station_num)
{
   if (!active_state[station_num]) {
      reportStateError("resetHemisphere", station_num);

      return(FALSE);
   }

   // Refer to pp 88 - 92, 3 SPACE USER's MANUAL for the default.
   float default_zenith[3];

   default_zenith[FSTK_X] = 1.0;
   default_zenith[FSTK_Y] = 0.0;
   default_zenith[FSTK_Z] = 0.0;

#if DEBUG
   cerr << "FASTRAK hemisphere reset.\n";
#endif

   return (setHemisphere(station_num, default_zenith));
}  // FastrakClass::resetHemisphere()


//------------------------------------------------------------------------
//   Function: getHemisphere
//    Returns:
// Parameters: station number and zenith vector
//    Summary: Get the hemisphere of the station.
//------------------------------------------------------------------------
void FastrakClass::getHemisphere(FSTK_stations station_num, float zenith[3])
{
   zenith[FSTK_X] = hemisphere[station_num][FSTK_X];
   zenith[FSTK_Y] = hemisphere[station_num][FSTK_Y];
   zenith[FSTK_Z] = hemisphere[station_num][FSTK_Z];
}  // FastrakClass::getHemisphere()
```

```
//-----------------------------------------------------------------------
//   Function: setUnits
//    Returns: TRUE if the operation is successful. Otherwais FALSE.
// Parameters:
//    Summary: Set the position measuring unit for the FASTRAK.
//-----------------------------------------------------------------------
int FastrakClass::setUnits(FSTK_units pos_units)
{
   // Construct the "U/u" command to set the unit for the FASTTRAC data.
   // Refer to pp 122 - 124, the 3 SPACE USER's MANUAL for details.
   char command[2];
   if (pos_units == FSTK_CENTIMETER)
      strcpy(command, "u\0");
   else if (pos_units == FSTK_INCH)
      strcpy(command, "U\0");
   else {
      cerr << "Error: invalid units specification " << pos_units << endl;
      return (FALSE);
   }

   int success = sendCommand(command, 1, "FastrakClass::setUnits");
   if (success) units = pos_units;

#if DEBUG
   cerr << "FASTRAK position units set.\n";
#endif

   return(success);
}  // end FastrakClass::setUnits


//-----------------------------------------------------------------------
//   Function: setDataTypes
//    Summary: Specify the requested data types for the station. Before
//           : data can be read from the FASTRAK, the types of the data
//           : needed must be specified. By default, position
//           : coordinates and Euler orientations are returned from each
//           : station.
// Parameters: station number, datatype mask
//    Returns: TRUE if the operation is successful. Otherwise, FALSE.
//-----------------------------------------------------------------------
int FastrakClass::setDataTypes(FSTK_stations station_num, short mask)
{
   if (!active_state[station_num]) {
      reportStateError("setDataTypes", station_num);

      return(FALSE);
   }

   int  success = TRUE;

   // The following piece of code is a critical section in multiprocess
   // mode.  When data record parameters are being updated, data records
```

```
// cannot be allowed to be read by getPacket() which runs in
// parallel with this method.
if ((parpoll_pid != -1) && (!param_set_flag)) {
   // entering the critical section
   if (uspsema(paramsema) == -1) {
           success = FALSE;
      report_syserr("getting semaphore",
                    "FastrakClass::setDataTypes");
   }
}

// Adjust the data record parameters.
int i, j;
fstk_packet_size -= datarec_size[station_num];

datarec_size[station_num] = FSTK_HEADER_SIZE;

for (i = 0; i < FSTK_NUM_DATATYPES; i++)
   datatype_start[station_num][i] = -1;

// Construct the "O" command to specify the type of data
// we want from the FASTRAK station.
// Refer to pp 97 - 111, the 3 SPACE USER's MANUAL for details.
char command[20];
sprintf(command, "O%d", station_num+1);

if (mask & FSTK_COORD_MASK) {
   strcat(command, ",2");
   datatype_start[station_num][FSTK_COORD_TYPE] =
      datarec_size[station_num];
   datarec_size[station_num] += FSTK_COORD_SIZE;
}

if (mask & FSTK_EULER_MASK)  {
   strcat(command, ",4");
   datatype_start[station_num][FSTK_EULER_TYPE] =
      datarec_size[station_num];
   datarec_size[station_num] += FSTK_EULER_SIZE;
}

if (mask & FSTK_XCOS_MASK) {
   strcat(command, ",5");
   datatype_start[station_num][FSTK_XCOS_TYPE] =
      datarec_size[station_num];
   datarec_size[station_num] += FSTK_XCOS_SIZE;
}
```

```
   if (mask & FSTK_YCOS_MASK) {
      strcat(command, ",6");
      datatype_start[station_num][FSTK_YCOS_TYPE] =
         datarec_size[station_num];
      datarec_size[station_num] += FSTK_YCOS_SIZE;
   }

   if (mask & FSTK_ZCOS_MASK) {
      strcat(command, ",7");
      datatype_start[station_num][FSTK_ZCOS_TYPE] =
         datarec_size[station_num];
      datarec_size[station_num] += FSTK_ZCOS_SIZE;
   }

   if (mask & FSTK_QUAT_MASK) {
      strcat(command, ",11");
      datatype_start[station_num][FSTK_QUAT_TYPE] =
         datarec_size[station_num];
      datarec_size[station_num] += FSTK_QUAT_SIZE;
   }

   if (mask & FSTK_16BIT_COORD_MASK) {
      strcat(command, ",18");
      datatype_start[station_num][FSTK_16BIT_COORD_TYPE] =
         datarec_size[station_num];
      datarec_size[station_num] += FSTK_16BIT_COORD_SIZE;
   }

   if (mask & FSTK_16BIT_EULER_MASK) {
      strcat(command, ",19");
      datatype_start[station_num][FSTK_16BIT_EULER_TYPE] =
         datarec_size[station_num];
      datarec_size[station_num] += FSTK_16BIT_EULER_SIZE;
   }

   if (mask & FSTK_16BIT_QUAT_MASK) {
      strcat(command, ",20");
      datatype_start[station_num][FSTK_16BIT_QUAT_TYPE] =
         datarec_size[station_num];
      datarec_size[station_num] += FSTK_16BIT_QUAT_SIZE;
   }

   if (mask & FSTK_CRLF_MASK) {
      strcat(command, ",1");
      datarec_size[station_num] += FSTK_CRLF_SIZE;
   }

   strcat(command, "\r");
```

```
   // recompute the maximum station data record size
   max_datarec_size = 0;
   for (i=0; i<FSTK_NUM_STATIONS; i++) {
      if (datarec_size[station_num] > max_datarec_size)
         max_datarec_size = datarec_size[station_num];
   }

#if DEBUG
   cerr << "** setDataTypes:\n"
        << "   The command: " << command << "\n"
        << "   Expecting station" << station_num+1
        << " to contain " << datarec_size[station_num]
        << " bytes." << endl;
#endif

   // Note that sendCommand() is not be used here because
   // adjusting data record params and sending the command must be in
   // the same critical section.
   if (write(port_fd, command, strlen(command)) == -1) {
      success = FALSE;
      report_syserr("sending FASTRAK O command",
                    "FastrakClass::setDataTypes");
   }

   datatype_mask[station_num] = mask;
   fstk_packet_size += datarec_size[station_num];

   for (i = station_num+1; i < FSTK_NUM_STATIONS; i++) {
      if (active_state[i]) {
         for (j = 0; j < FSTK_NUM_DATATYPES; j++)
            datatype_start[i][j] += datarec_size[station_num];
      }
   }

   data_ready_flag = FALSE; // See getPacket() for when it is set to TRUE.
   if ((parpoll_pid != -1) && (param_set_flag == FALSE))
      usvsema(paramsema);   // exiting the critical section

#if DEBUG
   cerr << "FASTRAK data type specified.\n";
   cerr << "Packet size set to " << fstk_packet_size << " bytes.\n";
#endif

   return(success);
}  // end FastrakClass::setDataTypes
```

```
//-----------------------------------------------------------------------
//   Function: setState
//    Summary: Set the sate of the station: TRUE (active) or FALSE
//           : (inactive).  Note  that this routine may seem to be more
//           : complex than necessary. The complexity is due to the need
//           : to handle the different requirements at and after the
//           : initialization stage. This routine can be simplified by
//           : spliting it into two separate ones. However, then the
//           : program becomes longer.
// Parameters: station number, state
//    Returns: TRUE if the operation is successful. Otherwise, FALSE.
//-----------------------------------------------------------------------
int FastrakClass::setState(FSTK_stations station_num, int active_flag)
{
   if (!active_setting[station_num])  {
      reportStateError("setState", station_num);
      return(FALSE);
   }

   int success = TRUE;

   // When fstk_packet_size = 0, the program is in the initialization stage,
   // where setState() is called from checkState(). Otherwise,
   // fstk_packet_size > 0. The initialization has finished and setState()
   // is called by the FASTRAK user.
   if ((fstk_packet_size == 0) ||
       (active_state[station_num] != active_flag)) {
      int i, j, active_station = -1, num_actives = 0;

      for (i = 0; i < FSTK_NUM_STATIONS; i++) {
         if (active_state[i]) {
            num_actives++;
            active_station = i;
         }
      }

      // Error! trying to deactivate the last remaining station.
      // At any time, at least one station must be active:
      if ((num_actives <= 1) &&
          (active_station == station_num) && !(active_flag)) {
         cerr << BELL << "Error in setting FASTRAK station"
              << station_num+1 << " state.\n"
              << "  in *FastrakClass.cc*FastrakClass::setState"
              << "  At least, one station must be active at any time."
              << endl;
         return(FALSE);
      }

      // Construct the "l" command to set the state of the station.
      // Refer to pp 128 - 131, the 3 SPACE USER's MANUAL for details.
      char command[10];
      int state_cmd = 0;
      if (active_flag) state_cmd = 1;
```

```
      sprintf(command, "l%d,%d\r", station_num+1, state_cmd);

#if DEBUG
      cerr << "** setState:\n"
           << "   The command: " << command << endl;
#endif

      // This is a critical section.
      if (parpoll_pid != -1) {
         if (uspsema(paramsema) == -1) {  // entering the critical section
            success = FALSE;
            report_syserr("getting semaphore", "FastrakClass::setState");
         }
      }

      // Note that sendCommand() is not be used here because
      // sending the command and updating data record params must be in
      // the same critical section.
      if (write(port_fd, command, strlen(command)) != -1) {
         active_state[station_num] = active_flag;

         // Update data record parameters when the change is from ACTIVE
         // to INACTIVE. On the other hand, when the change is from
         // INACTIVE to ACTIVE, setDataTypes() is called to update
         // record parameters.
         if (!active_flag && (fstk_packet_size > 0)) {
            fstk_packet_size -= datarec_size[station_num];

            for (i = station_num+1; i < FSTK_NUM_STATIONS; i++) {
               if (active_state[i]) {
                  for (j = 0; j < FSTK_NUM_DATATYPES; j++)
                     datatype_start[i][j] -= datarec_size[station_num];
               }
            }

            datarec_size[station_num] = 0;
            for (i = 0; i < FSTK_NUM_DATATYPES; i++) {
               datatype_start[station_num][i] = -1;
            }
         }
         else if (active_flag) {
            param_set_flag = TRUE;
            setDataTypes(station_num, FSTK_DEFAULT_MASK);
            param_set_flag = FALSE;
         }
      }
      else {
         success = FALSE;
         report_syserr("sending FASTRAK l command",
                       "FastrakClass::setState");
      }  // end if (write())

      data_ready_flag = FALSE;  // See getPacket() for when it is set to TRUE.
```

```
      // exiting the critical section
      if (parpoll_pid != -1) usvsema(paramsema);
   }

#if DEBUG
   cerr << "FASTRAK station" << station_num << " state set." << endl;
#endif

   return(success);
}  // end FastrakClass::setState()


//---------------------------------------------------------------------
//   Function: copyBuffer()
//    Summary:
// Parameters:
//    Returns:
//---------------------------------------------------------------------
void FastrakClass::copyBuffer()
{
   // Use uspsema(paramsema) instead of ussetlock(datalock) if the
   // object of this class is to be used in multiple processes.
   while (!data_ready_flag) ;
   if (ussetlock(datalock) == -1)   // locking for update
      report_syserr("getting lock", "FastrakClass::readData");

   for (int station_num=0; station_num<FSTK_NUM_STATIONS; station_num++) {
      memcpy(datarec[station_num], datarec_buf[station_num],
             datarec_size[station_num]);
   }

   // Use usvsema(paramsema) if the object is to be used in multiple
   // processes.
   usunsetlock(datalock);        // unlocking
}


//---------------------------------------------------------------------
//   Function: readData
//    Summary: Decompose the data packet in the current data buffer. For
//           : a succesful read, data_dest[] contains the required type
//           : of data from the specified station.  Note that (1)
//           : data_dest[] must be a 4-element array for quaternions;
//           : for the other types, it is a 3-element array; (2) old
//           : data can be reused if invalid data packets were read by
//           : getPacket().
// Parameters: station number, required data types
//    Returns: TRUE if the read is succesful. Otherwise, return FALSE.
//---------------------------------------------------------------------
int FastrakClass::readData(FSTK_stations station_num,
                           FSTK_datatypes data_type,
                           float data_dest[])
{
```

```
   if (checkReadError(station_num, "readData", data_type) == FALSE) {
      return(FALSE);
   }

/*
   // Use uspsema(paramsema) instead of ussetlock(datalock) if the
   // object of this class is to be used in multiple processes.
   while (!data_ready_flag) ;
   if (ussetlock(datalock) == -1)   // locking for update
      report_syserr("getting lock", "FastrakClass::readData");
 */
#if DEBUG
   cerr << "** readData:\n"
        << "   station" << station_num+1 << "\n"
        << "   record start = " << datarec[station_num]
        << "; record size = " << datarec_size[station_num]
        << "; total size = " << fstk_packet_size
        << "; data type start = "
        << datatype_start[station_num][data_type] << endl;

   debugData(datarec[station_num], datarec_size[station_num]);
#endif

   // the starting position of the type of data wanted in the buffer
   char* start_pos = datarec[station_num] +
                     datatype_start[station_num][data_type];

   switch (data_type) {
      case FSTK_COORD_TYPE:
      case FSTK_EULER_TYPE:
      case FSTK_XCOS_TYPE:
      case FSTK_YCOS_TYPE:
      case FSTK_ZCOS_TYPE:
         convertData(start_pos, 3, data_dest);
         break;

      case FSTK_QUAT_TYPE:
         convertData(start_pos, 4, data_dest);
         break;

      case FSTK_16BIT_COORD_TYPE:
         if (units == FSTK_CENTIMETER)
            convert16BITData(start_pos, 3,
                             FSTK_16BIT_TO_CM, data_dest);
         else
            convert16BITData(start_pos, 3,
                             FSTK_16BIT_TO_INCHES, data_dest);
         break;

      case FSTK_16BIT_EULER_TYPE:
         convert16BITData(start_pos, 3,
                          FSTK_16BIT_TO_DEGREES, data_dest);
         break;
```

```
      case FSTK_16BIT_QUAT_TYPE:
         convert16BITData(start_pos, 3,
                          FSTK_16BIT_TO_QUAT, data_dest);
         break;
   }

/*
   // Use usvsema(paramsema) if the object is to be used in multiple
   // processes.
   usunsetlock(datalock);        // unlocking
 */
   return(TRUE);
}  // end readData()


//-----------------------------------------------------------------------
//   Function: getHmatrix
//    Summary: Read a homogeneous transformation matrix. On a successful
//           : return, the upper left 3x3 submatrix of matrix[][]
//           : contains a tranformation matrix constructed from the
//           : X-cosin, Y-cosin and Z-cosin vectors of the station with
//           : X-consin in the first row, Y-cosin in the second, and
//           : Z-cosin in the third; if FSTK_COORD_TYPE has been chosen
//           : in setDataTypes, the fourth COLUMN will contain the
//           : position of the sensor wrt the transmitter, otherwise,
//           : it is filled with 0.  The fourth ROW is filled with 0's
//           : except that matrix[3][3] = 1.
// Parameters: station number
//    Returns: TRUE if the read is succesful. Otherwise, return FALSE.
//-----------------------------------------------------------------------
int FastrakClass::getHMatrix(FSTK_stations station_num,
                             float matrix[4][4])
{
   // the starting pos. of the types of data wanted in the data record
   char* start_pos;

   matrix[3][0] = matrix[3][1] = matrix[3][2] = 0;
   matrix[3][3] = 1;
/*
   // Use uspsema(paramsema) instead of ussetlock(datalock) if the
   // object of this class is to be used in multiple processes.
   while (!data_ready_flag) ;
   if (ussetlock(datalock) == -1)   // locking for update
      report_syserr("getting lock", "FastrakClass::getHmatrix");
 */

   // get the position vector if FSTK_COORD_TYPE has been specified
   if (datatype_mask[station_num]&FSTK_16BIT_COORD_MASK) {
      float pos[3];
      start_pos = datarec[station_num] +
         datatype_start[station_num][FSTK_16BIT_COORD_TYPE];
```

```
      if (units == FSTK_CENTIMETER)
         convert16BITData(start_pos, 3, FSTK_16BIT_TO_CM, pos);
      else
         convert16BITData(start_pos, 3, FSTK_16BIT_TO_INCHES, pos);

      for (int i=0; i<3; i++) matrix[i][3] = pos[i];
   }
   else if (datatype_mask[station_num]&FSTK_COORD_MASK) {
      float pos[3];
      start_pos = datarec[station_num] +
         datatype_start[station_num][FSTK_COORD_TYPE];
      convertData(start_pos, 3, pos);
      for (int i=0; i<3; i++) matrix[i][3] = pos[i];
   }
   else {
      matrix[0][3] = matrix[1][3] = matrix[2][3] = 0;
   }

   // get the rotation matrix one of three ways
   if (datatype_mask[station_num]&(FSTK_QUAT_MASK|FSTK_16BIT_QUAT_MASK)) {
      float quat[4];

      if (datatype_mask[station_num]&FSTK_16BIT_QUAT_MASK) {
         start_pos = datarec[station_num] +
            datatype_start[station_num][FSTK_16BIT_QUAT_TYPE];
         convert16BITData(start_pos, 4, FSTK_16BIT_TO_QUAT, quat);
      }
      else {
         start_pos = datarec[station_num] +
            datatype_start[station_num][FSTK_QUAT_TYPE];
         convertData(start_pos, 4, quat);
      }
//      usunsetlock(datalock);       // unlocking

      // compute the rotation matrix from the quaternion info
      float xx = quat[1]*quat[1];
      float yy = quat[2]*quat[2];
      float zz = quat[3]*quat[3];
      float xy = quat[1]*quat[2];
      float yz = quat[2]*quat[3];
      float xz = quat[1]*quat[3];
      float sx = quat[0]*quat[1];
      float sy = quat[0]*quat[2];
      float sz = quat[0]*quat[3];

      matrix[0][0] = 1.0 - 2.0*(yy + zz);
      matrix[0][1] = 2.0*(xy - sz);
      matrix[0][2] = 2.0*(xz + sy);

      matrix[1][0] = 2.0*(xy + sz);
      matrix[1][1] = 1.0 - 2.0*(xx + zz);
      matrix[1][2] = 2.0*(yz - sx);
```

```
      matrix[2][0] = 2.0*(xz - sy);
      matrix[2][1] = 2.0*(yz + sx);
      matrix[2][2] = 1.0 - 2.0*(xx + yy);
   }
   else if (datatype_mask[station_num]&
            (FSTK_EULER_MASK|FSTK_16BIT_EULER_MASK)) {
      float angles[3];

      if (datatype_mask[station_num]&FSTK_16BIT_EULER_MASK) {
         start_pos = datarec[station_num] +
            datatype_start[station_num][FSTK_16BIT_EULER_TYPE];
         convert16BITData(start_pos, 3, FSTK_16BIT_TO_DEGREES, angles);
      }
      else {
         start_pos = datarec[station_num] +
            datatype_start[station_num][FSTK_EULER_TYPE];
         convertData(start_pos, 3, angles);
      }
//      usunsetlock(datalock);       // unlocking

      // compute rotation matrix from the euler angle info
      angles[FSTK_AZ] *= DTOR;
      angles[FSTK_EL] *= DTOR;
      angles[FSTK_RO] *= DTOR;

      float ca = cos(angles[FSTK_AZ]);
      float sa = sin(angles[FSTK_AZ]);
      float ce = cos(angles[FSTK_EL]);
      float se = sin(angles[FSTK_EL]);
      float cr = cos(angles[FSTK_RO]);
      float sr = sin(angles[FSTK_RO]);

      float sesr = se*sr;
      float secr = se*cr;

      matrix[0][0] = ca*ce;
      matrix[0][1] = ca*sesr - sa*cr;
      matrix[0][2] = ca*secr + sa*sr;

      matrix[1][0] = sa*ce;
      matrix[1][1] = sa*sesr + ca*cr;
      matrix[1][2] = sa*secr - ca*sr;

      matrix[2][0] = -se;
      matrix[2][1] = ce*sr;
      matrix[2][2] = ce*cr;
   }
```

```
   else if ((datatype_mask[station_num]&FSTK_XCOS_MASK) &&
            (datatype_mask[station_num]&FSTK_YCOS_MASK) &&
            (datatype_mask[station_num]&FSTK_ZCOS_MASK)) {
      for (int row = 0; row <3; row++) {
         start_pos = datarec[station_num] +
            datatype_start[station_num][row + FSTK_XCOS_TYPE];
         convertData(start_pos, 3, matrix[row]);
      }
//      usunsetlock(datalock);       // unlocking
   }
   else {
      cerr << "Error: no orientation information to build H-matrix\n";
//      usunsetlock(datalock);       // unlocking
      return (FALSE);
   }

   return(TRUE);
}  // end getHmatrix()


//-----------------------------------------------------------------------
//   Function: getPosOrient
//    Summary: Read the current position and orientation of the station
//           : together.  On a successful return, posit[] contains the
//           : position and orient[] contains the orientation.  The type
//           : of the orientation, euler-angle and quaternion, is
//           : determined by orient_type.  Note that if orient_type is
//           : FSTK_EULER_TYPE, orient is a 3-element array. Otherwise, it
//           : must be a 4-element array.
// Parameters: station number, type of orientation
//    Returns: TRUE if the read is succesful. Otherwise, return FALSE.
//-----------------------------------------------------------------------
int FastrakClass::getPosOrient(FSTK_stations station_num,
                               FSTK_datatypes orient_type,
                               float pos[3], float orient[])
{
   char* start_pos;

   if (checkReadError(station_num, "getPosOrient",
                      orient_type) == FALSE) {
         return(FALSE);
   }
/*
   // Use uspsema(paramsema) instead of ussetlock(datalock) if the
   // object of this class is to be used in multiple processes.
   while (!data_ready_flag) ;
   if (ussetlock(datalock) == -1)   // locking for update
      report_syserr("getting lock", "FastrakClass::getPosOrient");
 */
   // get the position vector
   if (datatype_mask[station_num]&FSTK_16BIT_COORD_MASK) {
      start_pos = datarec[station_num] +
         datatype_start[station_num][FSTK_16BIT_COORD_TYPE];
```

```
      if (units == FSTK_CENTIMETER)
         convert16BITData(start_pos, 3, FSTK_16BIT_TO_CM, pos);
      else
         convert16BITData(start_pos, 3, FSTK_16BIT_TO_INCHES, pos);
   }
   else if (datatype_mask[station_num]&FSTK_COORD_MASK) {
      start_pos = datarec[station_num] +
         datatype_start[station_num][FSTK_COORD_TYPE];
      convertData(start_pos, 3, pos);
   }
   else {
      cerr << "Error: no position type selected in "
           << "FastrackClass::getPosOrient\n";
      return(FALSE);
   }

   // get orientation vector
   start_pos = datarec[station_num] +
                    datatype_start[station_num][orient_type];

   switch (orient_type) {
      case FSTK_EULER_TYPE:
         convertData(start_pos, 3, orient);
         break;

      case FSTK_QUAT_TYPE:
         convertData(start_pos, 4, orient);
         break;

      case FSTK_16BIT_EULER_TYPE:
         convert16BITData(start_pos, 3,
                          FSTK_16BIT_TO_DEGREES, orient);
         break;

      case FSTK_16BIT_QUAT_TYPE:
         convert16BITData(start_pos, 3,
                          FSTK_16BIT_TO_QUAT, orient);
         break;
      default:
         cerr << "Error: invalid orientation type specified in "
              << "FastrackClass::getPosOrient\n";
         return(FALSE);
   }
/*
   usunsetlock(datalock);       // unlocking
 */
   return(TRUE);
}  // end getPosOrient()
```

APPENDIX C: FASTRAK CONFIGURATION FILE

fastrak.dat **1**

```
# ******************************************************
# FILENAME:  fastrak.dat
# PURPOSE: configuration file for Body class using
#        : four fastrak sensors
# AUTHOR:  P F Skopowski
# DATE: 1 Jul 96
# COMMENTS:
# The file format:
#   a). A line starting with a '#' is a comment line.
#   b). Each line must not contain more than 255 characters.
#   c). Maintain the order of the parameters (i.e., the station
#       parameters, hemisphere and alignment, must be the last
#       part of the file).
# ******************************************************


# ================ Parameters for the FastrakClass ==================
# the serial port name for the FASTRAK
PORT: /dev/ttyd2

# Which station do you want to work with?
# A station can set to be active or inactive by the software. Only
# active stations return data. Only the station with its hardware
# switch set on can be set to be active by the software.
# Set the corresponding bit to 1 if you want the station to be active.
# Note that at any time, at least one station must be active.
WANTED_STATIONS: 1 1 1 1

# the parameters for the hemisphere and alignment of each station

# These following parameters must be the last part of the file.
# The parameters for a station do not have to be specified here.
# If they are not specified, the default values of the FASTRAK are used.
# The STATION#_PARAM line and the four parameter lines following it must
# immediately follow one another. There can be no comment lines among them.

# the hemisphere and alignment of station 1
STATION1_PARAM:
  hemisphere:     0  0 -1
  origin:         0  0  0
  x_point:        0 -1  0
  y_point:       -1  0  0

# the hemisphere and alignment of station 2
STATION2_PARAM:
  hemisphere:  0  0 -1
  origin:      0  0  0
  x_point:     0 -1  0
  y_point:    -1  0  0
```

```
# the hemisphere and alignment of station 3
STATION3_PARAM:
  hemisphere:  0  0 -1
  origin:      0  0  0
  x_point:     0 -1  0
  y_point:    -1  0  0

# the hemisphere and alignment of station 4
STATION4_PARAM:
  hemisphere:  0  0 -1
  origin:      0  0  0
  x_point:        0 -1  0
  y_point:    -1  0  0
```

APPENDIX D: POSITION TRACKING SOFTWARE

body.h **1**

```
// ******************************************************
// FILENAME:  body.h
// PURPOSE:  declarations for the Body class
//        :  uses position tracking technique
// AUTHOR:  P F Skopowski
// DATE: 1 Aug 96
// COMMENTS: definition of the Body class
// ******************************************************

#ifndef BODY_H
#define BODY_H

#define PF_CPLUSPLUS_API 0
#include <Performer/pf.h>
#include “upperbody.h”
#include “lowerbody.h”
#include “FastrakClass.h”



class Body
{
   private:
      Upperbody upperbody;
      Lowerbody lowerbody;

      int valid;

      FastrakClass *fastrak_unit;

      FSTK_stations torso_sensor;
      FSTK_stations upperarm_sensor;
      FSTK_stations lowerarm_sensor;
      FSTK_stations hand_sensor;

      // Fastrak related coordinate systems
      pfMatrix H_tx_to_ts, H_tx_to_uas, H_tx_to_las, H_tx_to_hs;

      // Calibration matrices
      pfMatrix H_ts_to_link3, H_ts_to_link6, H_uas_to_link20;
      pfMatrix H_las_to_link21, H_hs_to_link24;
      pfMatrix H_ts_to_posit16, H_uas_to_posit18;

      // Graphical model related coordinate systems
      pfMatrix H3, H6, H20, H21, H24;
      pfMatrix H_posit16, H_posit18;
```

```
      // Body part lengths
      float spine_shoulder_length, uarm_length, larm_length, hand_length;

      void outputHMatrix(pfMatrix H_mat);


   public:
      Body(const char *cfg_filename);

      ~Body();

      void rotate (double *);

      void rotate_increment (double *);

      void draw();

      void reset();

      int  exists() { return valid; }

      void get_all_inputs();

      int  calibrate();

      int  set_joint_angles();

      int calculate_joint_angles(double *);

      int set_link_length(int, float);

      int set_joint_displacement(int, float);

};


#endif
```

```
// *******************************************************
// FILENAME:  body.cc
// PURPOSE:  functions for the Body class
//        :  position tracking technique
// AUTHOR:  P F Skopowski
// DATE: 1 Aug 96
// COMMENTS: functions for the Body class
// *******************************************************

#include <math.h>
#include <iostream.h>
#include “body.h”

//-------------------------------------------------------
// Function: Body(const char *config_filename)
// Purpose: constructor of the body type
//        : creates and initializes FastrakClass object
//        : uses fastrak.dat configuration file
// Returns: body class object
//-------------------------------------------------------
Body::Body(const char *config_filename)
{
   valid = FALSE;

   fastrak_unit = NULL;

   // open configuration file
   ifstream config_fileobj(config_filename);
   if (!config_fileobj) {
      cerr << “Error: opening configuration file: “
           << config_filename << endl;
      return;
   }


   // initialize matrices
   pfMakeIdentMat(H_tx_to_ts);
   pfMakeIdentMat(H_tx_to_uas);
   pfMakeIdentMat(H_tx_to_las);
   pfMakeIdentMat(H_tx_to_hs);
   pfMakeIdentMat(H_ts_to_link3);
   pfMakeIdentMat(H_ts_to_link6);
   pfMakeIdentMat(H_uas_to_link20);
   pfMakeIdentMat(H_las_to_link21);
   pfMakeIdentMat(H_hs_to_link24);
   pfMakeIdentMat(H_ts_to_posit16);
   pfMakeIdentMat(H_uas_to_posit18);
   pfMakeIdentMat(H3);
   pfMakeIdentMat(H6);
   pfMakeIdentMat(H20);
   pfMakeIdentMat(H21);
   pfMakeIdentMat(H24);
```

```
   //initialize Fastrak
   fastrak_unit = new FastrakClass(config_fileobj);

   if (fastrak_unit->exists()) {
      if (fastrak_unit->getState(FSTK_STATION1))
         torso_sensor = FSTK_STATION1;
      if (fastrak_unit->getState(FSTK_STATION2))
         upperarm_sensor  = FSTK_STATION2;
      if (fastrak_unit->getState(FSTK_STATION3))
         lowerarm_sensor = FSTK_STATION3;
      if (fastrak_unit->getState(FSTK_STATION4))
         hand_sensor = FSTK_STATION4;

      valid = TRUE;
   }

}


//-------------------------------------------------------
// Function: ~Body()
// Purpose: destructor of the body type
//-------------------------------------------------------
Body::~Body()
{
   if ((fastrak_unit != NULL) && (fastrak_unit->exists())) {
      delete fastrak_unit;
      fastrak_unit = NULL;
   }
}


//-------------------------------------------------------
// Function: rotate (double *angles)
// Purpose: set upperbody joint angles
//        : uses the passed in array of values
//-------------------------------------------------------
void Body::rotate (double *angles)
{
    upperbody.rotate(angles);
}


//-------------------------------------------------------
// Function: rotate_increment (double *increment_angles)
// Purpose: increment upperbody joint angles
//        : uses the passed in array of values
//-------------------------------------------------------
void Body::rotate_increment (double *increment_angles)
{
    upperbody.rotate_increment(increment_angles);
}
```

```
//---------------------------------------------------------
// Function: draw()
// Purpose: draw the body in the proper position
//---------------------------------------------------------
void Body::draw()
{
    glPushMatrix();
    upperbody.draw();
    glPopMatrix();
    lowerbody.draw();
}


//---------------------------------------------------------
// Function: reset()
// Purpose: reset upperbody joint angles
//---------------------------------------------------------
void Body::reset()
{
    upperbody.reset();
}


//---------------------------------------------------------
// Function: set_link_length(int link, float length)
// Purpose: set  a  specified link's length
//        : used to size the link to the user
// Returns: TRUE if successful
//---------------------------------------------------------
int Body::set_link_length(int link, float length)
{
    if(upperbody.set_link_length(link, length)){
       return TRUE;
    }
    return FALSE;
}


//---------------------------------------------------------
// Function: set_joint_displacement(int link, float length)
// Purpose: set  a  specified link's joint displacement
//        : used to size the link to the user
// Returns: TRUE if successful
//---------------------------------------------------------
int Body::set_joint_displacement(int link, float length)
{
    if(upperbody.set_joint_displacement(link, length)){
       return TRUE;
    }
    return FALSE;
}
```

```
//-------------------------------------------------------
// Function: get_all_inputs()
// Purpose: get inputs from the fastrak trackers
//        : called to copy latest sample from second buffer
//        : implemented for double buffering to reduce
//        : lock overhead
//        : called once at the beginning of each frame
// Comment: original interface design by Scott McMillan
//-------------------------------------------------------
void Body::get_all_inputs()
{

   if (fastrak_unit->exists()) {
      fastrak_unit->copyBuffer();

      fastrak_unit->getHMatrix(torso_sensor, H_tx_to_ts);
      fastrak_unit->getHMatrix(upperarm_sensor, H_tx_to_uas);
      fastrak_unit->getHMatrix(lowerarm_sensor,  H_tx_to_las);
      fastrak_unit->getHMatrix(hand_sensor, H_tx_to_hs);




   }
}


//-------------------------------------------------------
// Function: output
// Purpose: output homogeneous transformation matrix (4x4)
//-------------------------------------------------------
void Body::outputHMatrix(pfMatrix Hmat)
{
   for (int i=0; i<4; i++)
      printf(" %6.3f %6.3f %6.3f %6.3f\n",
             Hmat[i][0], Hmat[i][1], Hmat[i][2], Hmat[i][3]);
   printf("\n");
}



//-------------------------------------------------------
// Function: set_joint_angles()
// Purpose: Set the body's joint angles using fastrak data
// Returns: TRUE if successful
//-------------------------------------------------------
int Body::set_joint_angles()
{
   int valid = FALSE;

   double angles[25];
```

```
   for (int i = 0; i < 25; i++){
      angles[i] = 0.0;
   }

   valid = calculate_joint_angles(angles);

   if (valid){
      rotate(angles);
   }

   return valid;
}



//-------------------------------------------------------
// Function: calculate_joint_angles(double *)
// Purpose: calculate inverse kinematics
//        : return the joint angles
//        : get_all_inputs must run first to update data
// Returns: TRUE if successful
//-------------------------------------------------------
int Body::calculate_joint_angles(double *angles)
{
   int valid = FALSE;
   double theta1 = 0.0;
   double theta2 = 0.0;
   double theta3 = 0.0;
   double theta18 = 0.0;
   double theta19 = 0.0;
   double theta20 = 0.0;
   double theta21 = 0.0;
   double theta22 = 0.0;
   double theta23 = 0.0;
   double theta24 = 0.0;


   const double deg_to_rad = .017453292519943295;

   if (fastrak_unit->exists()) {
      // must call get_all_inputs() first
      valid = TRUE;

      // convert reported data using calibration matrices
      pfMultMat(H3, H_tx_to_ts, H_ts_to_link3);//using calib matrix

      //pfMatrix H_ts_desired = {{ 0.0,  0.0, -1.0, 0.0},
      //                         { 0.0, -1.0,  0.0, 0.0},
      //                         {-1.0,  0.0,  0.0, 0.0},
      //                         { 0.0,  0.0,  0.0, 1.0}};//not using data
      //pfCopyMat (H3, H_ts_desired);//not using calib

      pfMultMat(H20, H_tx_to_uas, H_uas_to_link20); //using calib matrix
```

```
//*************investigate offset tracking***********************

pfMatrix  H3_inv;

pfMultMat(H6, H_tx_to_ts, H_ts_to_link6);//using calib matrix

pfMultMat(H_posit16, H_tx_to_ts, H_ts_to_posit16);

pfMultMat(H_posit18, H_tx_to_uas, H_uas_to_posit18);

pfMatrix temp1, temp2;
pfMakeIdentMat(temp1);
pfMakeIdentMat(temp2);
temp1[0][3] = H_posit18[0][3] - H_posit16[0][3];
temp1[1][3] = H_posit18[1][3] - H_posit16[1][3];
temp1[2][3] = H_posit18[2][3] - H_posit16[2][3];

// now convert the vector to a torso coord system
pfMatrix R6, R6_inv;
pfMakeIdentMat(R6);
pfCopyMat(R6, H6);
pfSetMatCol(R6, 3, 0.0, 0.0, 0.0, 1.0);
pfTransposeMat(R6_inv, R6);
pfMultMat(temp2, R6_inv, temp1);

float theta16 = atan2(temp2[2][3], temp2[0][3]);
if (theta16 < 0.0){
   theta16 += 6.283185307;
}

float length = sqrt(temp2[0][3] * temp2[0][3] +
                                  temp2[2][3] * temp2[2][3]);
float theta17 = atan2(-temp2[1][3], length);

//*************************************************************

pfMultMat(H21, H_tx_to_las, H_las_to_link21);

pfMultMat(H24, H_tx_to_hs, H_hs_to_link24);

// get the data from H3
double a3 = H3[2][0];
double b3 = H3[2][1];
double c3 = H3[2][2];
double c2 = H3[1][2];
double c1 = H3[0][2];

// compute the sin of theta2
double sin_theta2 = sqrt(c1 * c1 + c2 * c2);
```

```
      // check for zero
      if (sin_theta2 < 0.001){
         sin_theta2 = 0.001;
      }

      // set the sign of the answer
      if (c1 < 0.0){
         sin_theta2 *= -1.0;
      }

      // compute the angles
      theta2 = atan2(sin_theta2, -c3);
      theta3 = atan2(-b3/sin_theta2, a3/sin_theta2);
      theta1 = atan2(c2/sin_theta2, c1/sin_theta2);


      // compute T_17_to_20
      pfMatrix T_17_to_20, H_temp;
      pfMatrix T_17_to_3 = {{ 0.0, -1.0,  0.0,  0.75},
                            { 1.0,  0.0,  0.0,  4.0 },
                            { 0.0,  0.0,  1.0,  0.0 },
                            { 0.0,  0.0,  0.0,  1.0 }};

      pfInvertFullMat(H3_inv, H3);

      pfMultMat(H_temp, H3_inv, H20);
      pfMultMat(T_17_to_20, T_17_to_3, H_temp);

      // get the data from T_17_to_20
      double a2 = T_17_to_20[1][0];
      double b2 = T_17_to_20[1][1];
             c3 = T_17_to_20[2][2];
             c2 = T_17_to_20[1][2];
             c1 = T_17_to_20[0][2];

      // compute the sin of theta19
      double sin_theta19 = sqrt(a2 * a2 + b2 * b2);

      // check for zero
      if (sin_theta19 < 0.001){
         sin_theta19 = 0.001;
      }

      // set the sign of the answer
      if (c1 < 0.0){
         sin_theta2 *= -1.0;
      }

      // compute the angles
      theta19 = atan2(sin_theta19, c2);
      theta20 = atan2(b2/sin_theta19, -a2/sin_theta19);
      theta18 = atan2(c3/sin_theta19, c1/sin_theta19);
```

```
      // compute T_20_to_21
      pfMatrix T_20_to_21, H20_inv;

      pfInvertFullMat(H20_inv, H20);

      pfMultMat(T_20_to_21, H20_inv, H21);

      // get the data from T_20_to_21
      a3 = T_20_to_21[2][0];
      b3 = T_20_to_21[2][1];

      // compute the angle
      theta21 = atan2(a3, b3);

      pfMatrix T_21_to_24, H21_inv;

      pfInvertFullMat(H21_inv, H21);

      pfMultMat(T_21_to_24, H21_inv, H24);

      // get the data from H24
      a3 = T_21_to_24[2][0];
      b3 = T_21_to_24[2][1];
      c3 = T_21_to_24[2][2];
      c2 = T_21_to_24[1][2];
      c1 = T_21_to_24[0][2];

      // compute the sin of theta23
      double sin_theta23 = -sqrt(a3 * a3 + b3 * b3);

      // check for zero
      if (sin_theta23 > -0.001){
         sin_theta23 = -0.001;
      }

      // compute the angles
      theta23 = atan2(sin_theta23, -c3);
      theta24 = atan2(b3/sin_theta23, -a3/sin_theta23);
      theta22 = atan2(-c2/sin_theta23, -c1/sin_theta23);

      // convert all angles to degrees
      theta1 /= deg_to_rad;
      theta2 /= deg_to_rad;
      theta3 /= deg_to_rad;
      theta16 /= deg_to_rad;
      theta17 /= deg_to_rad;
      theta18 /= deg_to_rad;
      theta19 /= deg_to_rad;
      theta20 /= deg_to_rad;
      theta21 /= deg_to_rad;
      theta22 /= deg_to_rad;
      theta23 /= deg_to_rad;
      theta24 /= deg_to_rad;
```

```
      angles[1]  =  theta1;
      angles[2]  =  theta2;
      angles[3]  =  theta3;
      angles[4]  =   90.0;
      angles[5]  =   90.0;
      angles[6]  =  180.0;
      angles[7]  =    0.0;
      angles[8]  =    0.0;
      angles[9]  =   90.0;
      angles[10] =   90.0;
      angles[11] =    0.0;
      angles[12] =    0.0;
      angles[13] =    0.0;
      angles[14] =  -90.0;
      angles[15] =    0.0;
      angles[16] =  theta16;
      angles[17] =  theta17;
      angles[18] =   90.0;
      angles[19] =   90.0;
      angles[20] =    0.0;
      angles[21] =    0.0;
      angles[22] =    0.0;
      angles[23] =  -90.0;
      angles[24] =    0.0;
   }

   return valid;
}


//-------------------------------------------------------
// Function: calibrate()
// Purpose: size the upperbody model to the user
//        : calibrate the trackers
// Returns: TRUE if successful
//-------------------------------------------------------
int Body::calibrate()
{
   int valid = FALSE;

   pfMatrix H_torso_reported, H_uarm_reported;
   pfMatrix H_larm_reported, H_hand_reported;

   pfMakeIdentMat(H_torso_reported);
   pfMakeIdentMat(H_uarm_reported);
   pfMakeIdentMat(H_larm_reported);
   pfMakeIdentMat(H_hand_reported);

   if (fastrak_unit->exists()) {
      valid = TRUE;
      char str;
```

```
cerr << endl << “Calibrating sensor orientation in 3 seconds...” << endl;
cerr << “Press <Enter> to start count-down: “;
cin.get(str);
for (int i=0; i<3; i++) {
   sleep(1);
   cerr << (char) 7;
}

// this code allows the fastrak to do the calibration for torso
//float angles[3] = {90.0, -90.0, 180.0};
//fastrak_unit->setBoresight(torso_sensor, angles);

// get the data to compute the calibration matrices
fastrak_unit->copyBuffer();
fastrak_unit->getHMatrix(torso_sensor, H_torso_reported);
fastrak_unit->getHMatrix(upperarm_sensor, H_uarm_reported);
fastrak_unit->getHMatrix(lowerarm_sensor, H_larm_reported);
fastrak_unit->getHMatrix(hand_sensor,  H_hand_reported);



// compute the calibration matrices

// compute torso sensor calibration matrix
pfMatrix H_torso_reported_inv;

pfMatrix H_ts_desired = {{ 0.0,  0.0, -1.0, 0.0},
                         { 0.0, -1.0,  0.0, 0.0},
                         {-1.0,  0.0,  0.0, 0.0},
                         { 0.0,  0.0,  0.0, 1.0}};

pfInvertFullMat(H_torso_reported_inv, H_torso_reported);

pfMultMat(H_ts_to_link3, H_torso_reported_inv, H_ts_desired);

pfMatrix H_ts_desired2 = {{ 0.0,  0.0, -1.0, 0.0},
                          {-1.0,  0.0,  0.0, 0.0},
                          { 0.0,  1.0,  0.0, 0.0},
                          { 0.0,  0.0,  0.0, 1.0}};

pfMultMat(H_ts_to_link6, H_torso_reported_inv, H_ts_desired2);

//*************investigate offset tracking***********************

// compute torso sensor calibration matrix for posit16 tracking
// some necessary matrices
pfMatrix R_tx_to_ts, R_ts_to_tx;

pfCopyMat(R_tx_to_ts, H_torso_reported);

// set posit col to zero to work with rotation matrix only
pfSetMatCol(R_tx_to_ts, 3, 0.0, 0.0, 0.0, 1.0);
```

```
      // get the inverse rotation matrix
      pfTransposeMat(R_ts_to_tx, R_tx_to_ts);

      // determine suitable offsets from ts and uas position data
      float x_offset = H_uarm_reported[0][3] - H_torso_reported[0][3];
      float y_offset = 8.0;
      float z_offset = 0.0;

      // the offset from ts to rclav in world coordinates
      pfMatrix P_offset_ts_to_rclav  = {{ 1.0,  0.0,  0.0,   x_offset},
                                       { 0.0,  1.0,  0.0,   y_offset},
                                       { 0.0,  0.0,  1.0,   z_offset},
                                       { 0.0,  0.0,  0.0,   1.0}};
      pfMatrix temp;
      pfMultMat(temp, R_ts_to_tx, P_offset_ts_to_rclav);
      pfSetMatCol(H_ts_to_posit16, 3, temp[0][3], temp[1][3], temp[2][3], 1.0);


      // compute upper arm sensor calibration matrix
      pfMatrix H_uarm_reported_inv;

      pfMatrix H_uarm_desired = {{ 0.0,  0.0, -1.0, 0.0},
                                 { 0.0,  1.0,  0.0, 0.0},
                                 { 1.0,  0.0,  0.0, 0.0},
                                 { 0.0,  0.0,  0.0, 1.0}};

      pfInvertFullMat(H_uarm_reported_inv, H_uarm_reported);

      pfMultMat(H_uas_to_link20, H_uarm_reported_inv, H_uarm_desired);

      // compute upper arm sensor calibration matrix for posit18 tracking

      // some necessary matrices
      pfMatrix R_tx_to_uas, R_uas_to_tx;

      pfCopyMat(R_tx_to_uas, H_uarm_reported);

      // set posit col to zero to work with rotation matrix only
      pfSetMatCol(R_tx_to_uas, 3, 0.0, 0.0, 0.0, 1.0);

      // get the inverse rotation matrix
      pfTransposeMat(R_uas_to_tx, R_tx_to_uas);

      // determine suitable offsets from uas and ts position data
      x_offset = 0.0;
      y_offset = - (fabs(H_torso_reported[1][3] - H_uarm_reported[1][3]) -
22.0);
      z_offset = - fabs(H_torso_reported[2][3] - H_uarm_reported[2][3]);
```

```
      // the offset from uas to shoulder in world coordinates
      pfMatrix P_offset_uas_to_shoulder = {{ 1.0,  0.0,  0.0,   x_offset},
                                           { 0.0,  1.0,  0.0,   y_offset},
                                           { 0.0,  0.0,  1.0,   z_offset},
                                           { 0.0,  0.0,  0.0,   1.0}};

      pfMultMat(temp, R_uas_to_tx, P_offset_uas_to_shoulder);
      pfSetMatCol(H_uas_to_posit18, 3, temp[0][3], temp[1][3], temp[2][3], 1.0);

      //************************************************************

      // compute lower arm sensor calibration matrix
      pfMatrix H_larm_reported_inv;

      pfMatrix H_larm_desired = {{ 0.0, -1.0,  0.0, 0.0},
                                 { 0.0,  0.0, -1.0, 0.0},
                                 { 1.0,  0.0,  0.0, 0.0},
                                 { 0.0,  0.0,  0.0, 1.0}};

      pfInvertFullMat(H_larm_reported_inv, H_larm_reported);

      pfMultMat(H_las_to_link21, H_larm_reported_inv, H_larm_desired);

      // compute hand sensor calibration matrix
      pfMatrix H_hand_reported_inv;

      pfMatrix H_hand_desired = {{ 0.0,  1.0,  0.0, 0.0},
                                 {-1.0,  0.0,  0.0, 0.0},
                                 { 0.0,  0.0,  1.0, 0.0},
                                 { 0.0,  0.0,  0.0, 1.0}};

      pfInvertFullMat(H_hand_reported_inv, H_hand_reported);

      pfMultMat(H_hs_to_link24, H_hand_reported_inv, H_hand_desired);


      // set upperbody dimensions to that of the user
      set_link_length(3, 21.0);
      set_link_length(6, 8.0);
      set_joint_displacement(16, 26.0);
      set_link_length(17, 14.0);
      set_link_length(8, 14.0);
      set_link_length(20, 28.0);
      set_link_length(11, 28.0);
      set_link_length(21, 29.0);
      set_link_length(12, 29.0);
      set_link_length(24, 17.0);
      set_link_length(15, 17.0);
   }

   return valid;
}
```

APPENDIX E: DEMONSTRATION VIDEO

(SEE ACCOMPANYING VHS VIDEO TAPE:
“Immersive Articulation of the Human Upper Body
in a Virtual Environment
Appendix E: Demonstration Video”)

LIST OF REFERENCES

[ADVA96] Advanced Position Systems, Inc., "An RF Head Tracker," Defense Small Business Innovation Research (SBIR) Program Phase 1 Final Report, DoD Contract Number N00014-96-C-6206, Naval Research Laboratory, Washington, D. C., 19 October, 1996.

[ASCE95] Ascension Technology, http://www.oz.is/OZ/Misc/Ascension.html, Internet.

[BACH96a] Bachmann, E. R., McGhee, R. B., Whalen, R. H., Steven, R., Walker, R. G., Clynch, J. R. Healey, A. J., and Yun, X. P., "Evaluation of an Integrated GPS/INS System for Shallow-water AUV Navigation (SANS)," *Proceedings of the IEEE Symposium on Autonomous Underwater Vehicle Technology, AUV '96,* Monterey, CA, 3-6 June, 1996, pp. 268-275.

[BACH96b] Bachmann, E. R., *CS4920 Lecture Notes: Quaternion Attitude Filter,* Naval Postgraduate School, Monterey, CA, November 1996.

[BADL93a] Badler, N. I., Phillips, C. B. and Webber, B. L., *Simulating Humans*: *Computer Graphics Animation and Control,* Oxford University Press, New York, 1993.

[BADL93b] Badler, N. I., Hollick, M. J. and Granieri, J. P., "Real-Time Control of a Virtual Human Using Minimal Sensors," *Presence: Teleoperators and Virtual Environments,* Winter 1993, Volume 2, Number 1, pp. 82-86.

[BIBL95] Bible, Steven R., Zyda, Michael, Brutzman, Don, "Using Spread-Spectrum Ranging Techniques for Position Tracking in a Virtual Environment," Second IEEE Workshop on Networked Realities, Boston, MA, October 26-28 1995.

[CANT95] Canterbury, Michael, *An Automated Approach to Distributed Interactive Simulation (DIS) Protocol Entity Development*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1995.

[COOK92] Cooke, Joseph M., Zyda, Michael J., Pratt, David R., and McGhee, Robert B., "NPSNET: Flight Simulation Dynamic Modeling Using Quaternions," *Presence*, Fall 1992, Volume 1, Number 4, pp. 405-420.

[CRAI89] Craig, J., *Introduction to Robotics: Mechanics and Control*, Second Edition, Addison-Wesley Publishing Company, Inc., Menlo Park, California, 1989.

[DAVI93] Davidson, Sandra L., *An Experimental Comparison of CLOS and C++ Implementations of an Object-Oriented Graphical Simulation of Walking Robot Kinematics*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1993.

[DURL95] Durlach, N. I. and Mavor, A. S., National Research Council, *Virtual Reality: Scientific and Technological Challenges*, National Academy Press, Washington, D.C., 1995, pp. 188-204, 306-317.

[FOXL96] Foxlin, Eric, "Inertial Head-Tracker Sensor Fusion by a Complementary Separate-Bias Kalman Filter," *Procedings of VRAIS '96*, IEEE, 1996, pps 185-194.

[FREY96] Frey, William, III, "Application of Inertial Sensors and Flux-Gate Magnetometer to Real-Time Human Body Motion Capture," Master's Thesis, Naval Postgraduate School, Monterey, CA, September 1996.

[GRAN95] Granieri, J. P. and Badler, N. I.,"Simulating Humans in VR," *Virtual Reality Applications*, Academic Press, ISBN 0-12-227755-4, 1995, pp. 253-269.

[GUBI74] Gubina, Ferdinand, Hemami, Hooshang, and McGhee, Robert B., "On the Dynamic Stability of Biped Locomotion," *IEEE Transactions on Biomedical Engineering*, Vol. BME-21, No. 2, March 1974.

[HODG95] Hodgins, J. K., Wooten, W. L., Brogan, D. C., O'Brien, J. F., "Animating Human Athletics," *Proceedings of SIGGRAPH '95*, Los Angeles, CA, August 6-11, In *Computer Graphics*, 1995, pp 71-78.

[INTE96] InterSense Incorporated, *IS-300 Series User's Guide*, Cambridge, MA, 1996.

[KLEI83] Klein, C. A. and Huang, C., "Review of Pseudoinverse Control for Use with Kinematically Redundant Manipulators," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-13, No. 3, March/April 1983.

[KOOZ80] Koozekanani, S. H., Stockwell, C. W., McGhee, R. B., and Firoozmand, F., "On the Role of Dynamic Models in Quantitative Posturography," *IEEE Transactions on Biomedical Engineering*, October 1980, Volume BME-27, Number 10, pp. 605-609.

[KOOZ83] Koozekanani, S. H., Barin, K., McGhee, R. B., and Chang, H. T., "A Recursive Free-Body Approach to Computer Simulation of Human Postural Dynamics," *IEEE Transactions on Biomedical Engineering*, December 1983, Volume BME-30, Number 12, pp. 787-792.

[KWAK90] Kwak, S. H. and McGhee, R. B., “Rule-based Motion Coordination for a Hexapod Walking Machine,” *Advanced Robotics*, 1990 Volume 4, Number 3, pp. 263-282.

[LIPM90] Lipman Electronic Engineering Ltd., *V-scope™ VS-100 Owner’s Guide Rev. 1.3*, Cat. No. 050-32-002, 1990.

[LIVI96] Livingston, M. A. and State, A.,“Magnetic Tracker Calibration ofr Improved Augmented Reality Registration,” University of North Carolina, Chapel Hill, NC, submitted to *Presence*, 1996.

[LOGS92] Logston, T., *The Navstar Global Positioning System*, Van Nostrand Reinhold, New York, 1992.

[MCGH79] McGhee, R. B., Koozekanani, S. H., Weimer, F. C., and Rahmani, S., “Dynamic Modelling of Human Locomotion,” *Proceedings of IEEE Joint Automatic Control Conference*, Denver CO, 1979, pp. 405-413.

[MCMI94] McMillan, Scott, *Computational Dynamics for Robotic Systems on Land and Under Water*, Ph.D. Dissertation, Ohio State University, 1994.

[MCMI95] McMillan, S, Orin, D. E., and McGhee, R. B., “DynaMechs: An Object Oriented Software Package for Efficient Dynamic Simulation of URVs,” *Underwater Robotic Vehicles: Design and Control*, TSI Press, Albuquerque, NM, 1995, pp 73-98.

[MCMI96a] McMillan, Scott, “A Computational Framework for Simulation of Underwater Robotic Vehicle Systems,” *Autonomous Robots*, 3, pps 253-268, Kluwer Academic Publishers, Norwell, MA, 1996.

[MCMI96b] McMillan, Scott, “Upper Body Tracking Using the Polhemus Fastrak,” Technical Report NPSCS-96-002, Naval Postgraduate School, Monterey, CA, January 31, 1996.

[MEYE92] Meyer, K., Applewhite, H. L. and Biocca, F. A., “A Survey of Position Trackers,” *Presence: Teleoperators and Virtual Environments*, Spring, 1992, Volume 1, Number 2, pp. 173-200.

[POLH93] Polhemus, *3Space Fastrak User’s Manual Revision F*, OPM3609-002C, November 1993.

[PRAT94] Pratt, D. R., Barham, P. T., Locke, J., Zyda, M. J., Eastman, B., Moore, T., Biggers, K., Douglass, R., Jacobsen, S., Hollick, M., Granieri, J., Ko, H. and Badler, N. I., “Insertion of an Articulated Human into a Networked Virtual Environment,” *Proceedings of the Fifth Annual Conference on AI,*

Simulation and Planning in High Autonomy Systems: Distributed Interactive Simulation Environments, IEEE Computer Society Press, Gainesville, Florida, December 7-9,1994, pp. 84-90.

[PRAT95] Pratt, S. M., Pratt, D. R., Waldrop, M. S., Barham, P. T., Ehlert, J. F. and Chrislip, C. A., "Humans in a Large-Scale, Real Time, Networked Virtual Environments," submitted to *Presence*, 1996.

[SYSTRO] Systron Donner, Inc., *Systron Donner Model MP-GCCCQAAB MotionPak IMU*, Concord, CA.

[WALD95] Waldrop, Marianne S., *Real-time Articulation of the Upper Body for Simulated Humans in Virtual Environments*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1995.

[WATT92] Watt, A. and Watt, M., *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison-Wesley Publishing Company, Inc., New York, 1992, pp. 369-394.

[ZYDA92] Zyda, M. J., Pratt, D. R., Monahan, J. G. and Wilson, K. P., "NPSNET: Constructing a 3D Virtual World," *1992 Proceedings of Symposium on Interactive 3D Graphics*, pp. 147-156.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library.....2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Chairman, Code CS 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Robert B. McGhee, Professor.....2
   Computer Science Department Code CS/
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. John S. Falby, Senior Lecturer 2
   Computer Science Department Code CS/Fa
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Don Brutzman, Associate Professor 1
   Undersea Warfare, Code UW/Br
   Naval Postgraduate School
   Monterey, CA 93943-5000

7. Dr. Rudy Darken, Assistant Professor.....1
   Computer Science Department Code CS/DR
   Naval Postgraduate School
   Monterey, CA 93943-5000

8. Eric R. Bachmann, Lecturer 1
   Computer Science Department Code CS/Ba
   Naval Postgraduate School
   Monterey, CA 93943-5000

9. Director 2
   Marine Corps Research Center
   MCCDC, Code: C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

10. Director 1
    Studies and Analysis Division
    MCCDC, Code: C45
    3300 Russell Road
    Quantico, VA 22134-5130

11. Major Paul F. Skopowski 2
    4452 Majestic Lane
    Fairfax, VA 22033